Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matic Tribušon

Numerično reševanje valovne enačbe z grafično procesno enoto

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Moderne grafične procesne enote z močno paralelno arhitekturo predstavljajo zelo zmogljivo platformo za fizikalno modeliranje. Na primeru valovne enačbe primerjajte razvoj programske opreme in hitrost izvajanja simulacije v okolju nVidia CUDA in OpenCL. Preverite tudi kako se grafična procesna enota obnaša, ko jo poleg računanja uporabljamo še za izrisovanje rezultatov s knjižnico OpenGL.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matic Tribušon, z vpisno številko **63110333**, sem avtor diplomskega dela z naslovom:

Numerično reševanje valovne enačbe z grafično procesno enoto

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Uroša Lotriča,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 9. septembra 2014 Podpis avtorja:

Zahvaljujem se mentorju izr. prof. dr. Urošu Lotriču za vso strokovno pomoč, nasvete, popravke in mentorstvo pri izdelavi in pisanju diplomskega dela.
Zahvaljujem se vsem kolegom, ki so mi tekom študija pomagali z dodatnimi razlagami. Posebej bi se rad zahvalil kolegu Žigi Lesarju, ki je pri tem najbolj izstopal.
Zahvalil bi se rad tudi staršem za vso podporo pri dosedanjem študiju.

Kazalo

Seznam uporabljenih kratic

kratica	pomen
ALE	aritmetično logična enota
API	Application Programming Interface
CPE	centralna procesna enota
CUDA	Compute Unified Device Architecture
FPGA	Field-programmable gate array
GPE	grafična procesna enota
GPGPU	uporaba grafične kartice za namene računanja
OpenCL	Open Computing Language
OpenGL	Open Graphics Language
RK4	Runge-Kutta 4. reda
SM	Streaming Multiprocessor
SP	Streaming Processor

Povzetek

Cilj diplomske naloge je implementacija algoritma za numerično reševanje valovne enačbe na grafični procesni enoti. Diferencialno enačbo smo reševali z Eulerjevo metodo in metodo Runge-Kutta 4. reda. Metodi se razlikujeta po računski zahtevnosti, točnosti in numerični stabilnosti. Algoritma smo implementirali na platformah CUDA in OpenCL. Konkurenčni platformi smo med seboj primerjali in predstavili rezultate. Na koncu smo rezultate algoritmov v vsakem koraku vizualizirali z uporabo OpenGL in ocenili, kakšen vpliv na hitrost ima vizualizacija.
Rezultati potrjujejo hipotezo, da sta si platformi po zmogljivosti zelo podobni. CUDA je vendrale nekoliko hitrejša predvsem pri izračunih na nekoliko manjših matrikah, OpenCL pa je malce hitrejši pri večjih količinah podatkov. Vizualizacija v primerjavi z izračunom porabi ogromno časa. Zato ob vizualizaciji izbira platforme za programiranje na grafični procesni enoti ni ključnega pomena.

Ključne besede: valovna enačba, Eulerjeva metoda, metoda Runge-Kutta, CUDA, OpenCL, OpenGL.

Abstract

The aim of this thesis is implementation of algorithm for numerical solution of the wave equation on graphics processing unit. We used Euler and 4th order Runge-Kutta method. The methods differ in calculation complexity, numerical accuracy and numerical stability. Algorithms were implemented on CUDA and OpenCL platforms. Competitive platforms were compared with each other. Results of each step of the calculation were also visualized using OpenGL standard with the purpose of assessing the impact visualization has on time spent by algorithms.
Results confirm the hypothesis that the two GPGPU platforms are very similar in performance. CUDA is slightly faster on smaller matrices, OpenCL performs better on larger matrices. Visualization takes a lot of time compared to the calculation. Therefore, in the case of visualization, the choice of platform is not crucial.

Keywords: wave equation, Euler method, Runge-Kutta method, GPGPU, CUDA, OpenCL, OpenGL.

Poglavje 1

Uvod

V zadnjih nekaj desetletjih smo priča izjemno hitremu razvoju na področju tehnologije. Pred skoraj 50 leti je Gordon Moore z opazovanjem razvoja strojne opreme predvidel rast gostote tranzistorjev v digitalnih vezjih. To je seveda marsikomu dobro poznani Moorov zakon. Ta predvideva podvojitev gostote tranzistorjev približno vsaki dve leti. Obstaja več inačic in pogledov na Moorov zakon, enega izmed njih si bralec lahko ogleda v delu [1]. Kljub temu, da je bila to zelo drzna napoved, je dosedanji razvoj v veliki meri potrdil Moorova pričakovanja. Eksponentna rast gostote tranzistorjev nas je pripeljala do izjemno zmogljivih komponent kot so večjedrne centralne procesne enote in grafične procesne enote z več kot 5 milijardami tranzistorjev.

Moorov zakon se pogosto omenja v povezavi z napredkom centralnih procesnih enot. Čeprav se število tranzistorjev v CPE hitro povečuje, se sama frekvenca CPE v zadnjem desetletju ni veliko spremenila. Razlog za to je povsem fizikalne narave, saj se z višanjem frekvence zelo hitro veča tudi količina proizvedene toplote. Zato so se proizvajalci CPE začeli posluževati drugačne tehnike večanja zmogljivosti. Na en čip so namestili več jeder, ki lahko operacije izvajajo simultano. Prav zaradi tega se je v zadnjih nekaj letih zelo razmahnilo paralelno programiranje, ki ga potrebujemo za izkoriščanje novih CPE.

Tudi uporaba grafične procesne enote za računske namene je posebna vrsta paralelnega programiranja, le da je tukaj stopnja paralelnosti veliko večja. Novodobne

grafične procesne enote imajo več tisoč jeder in naloga programerja je, da jih čim bolj enakomerno obremeni.
Pred začetkom programiranja na grafični procesni enoti je vsak programer pred pomembno odločitvijo. Izbrati mora platformo, ki jo bo uporabil. Konkurenčni platformi CUDA in OpenCL sta si namreč zelo podobni. Če je cilj doseči neodvisnost od strojne opreme, je odgovor na dlani, saj CUDA deluje le na grafičnih procesnih enotah proizvajalca *nVidia*. V nasprotnem primeru pa je odločitev težja. Na spletu sicer lahko najdemo nekaj primerjav, navadno pa nimamo vpogleda v kodo programa, ki je bil izdelan na platformi CUDA ali OpenCL in uporabljen v primerjavi. Težavo v tem primeru predstavlja tudi hiter razvoj obeh platform, saj primerjave in testiranja hitro zastarajo ter lahko podvomimo v njihovo relevantnost. Vendarle pa lahko iz obsežnejših primerjav, kot je [2], vidimo, da je zmogljivost zelo odvisna od narave problema in algoritma, ki ga rešuje. V večini primerov je zmogljivost zelo podobna, ponekod pa se pojavljajo tudi večje razlike.

1.1 Cilji

V okviru diplomske naloge bomo predstavili in med seboj primerjali dve najbolj priljubljeni platformi za programiranje na grafičnih procesnih enotah. To sta CUDA in OpenCL. Med njima lahko potegnemo marsikatero vzporednico, najdemo pa lahko tudi pomembne razlike. Zaradi hude konkurence se obe platformi izredno hitro razvijata in zato se na kakšne starejše primerjave težko zanesemo.
Na obeh platformah smo implementirali reševanje valovne diferencialne enačbe z Eulerjevo metodo in metodo Runge-Kutta 4. reda. Platformi smo primerjali glede na čas, porabljen za izračun različnega števila korakov in različne velikosti matrik. Rezultati so predstavljeni v poglavju 7.
Ker se v realnem življenju navadno ne ustavimo le pri izračunu, pač pa želimo pridobljene podatke na nek način uporabiti, smo implementirali še grafični izris valovanja površine ter primerjali, koliko časa v primerjavi z izračunom porabi izris.

Poglavje 2

Uporaba grafične procesne enote za namene računanja

Grafična procesna enota ali grafična kartica je del vsakega računalnika. Skozi zgodovino lahko opazujemo razvoj različnih tipov grafičnih procesnih enot. Kot primer lahko vzamemo integrirane grafične procesne enote, ki navadno nimajo svojega pomnilnika, pač pa uporabljajo del glavnega pomnilnika računalnika in samostojne grafične procesne enote, ki predstavljajo zaključeno celoto in imajo lastni pomnilnik. Ne glede na tip grafične procesne enote pa lahko v zadnjem desetletju ali dveh opazimo izjemno hiter razvoj in napredek. Vzrokov je več. Eden izmed njih je zagotovo v hitrem napredku tehnologije, ki omogoča hiter razvoj tranzistorjev in ostalih sestavnih delov procesnih enot. K hitremu razvoju grafičnih procesnih enot po mojem mnenju veliko prispeva tudi velika priljubljenost in moč igračarske industrije, ki za ustvarjanje vedno bolj realne navidezne resničnosti potrebuje vse zmogljivejše grafične procesne enote.

Prav zaradi hitrega razvoja so grafične procesne enote v veliko pogledih prehitele zmožnosti CPE. Zato smo v zadnjem desetletju priča razmahu programiranja na grafičnih procesnih enotah oziroma uporabi GPE za namene računanja (t.i. general-purpose computing on graphics processing units).

Paralelno programiranje je prisotno že dolgo časa. Tudi večjedrne ali vsaj večnitne centralne procene enote poznamo že vsaj dve desetletji. Vendarle pa je programi-

ranje na grafičnih procesnih enotah zelo drugačno od paralelnega programiranja na centralnih procesnih enotah. Seveda si delita podobne koncepte, najdemo pa tudi veliko razlik. Glavna razlika je že v samem problemu, ki mora biti za učinkovito izvedbo na grafični procesni enoti zelo paralelen, saj moramo na GPE izkoristiti veliko več jeder kot na centralni procesni enoti. Druga pomembna razlika je pomnilnik. Pri paralelnem programiranju na centralnih procesnih enotah se način dostopa do pomnilnika ne razlikuje od navadnega programiranja. Seveda je potrebno paziti na sočasne dostope in dostopanje v pravem zaporedju, vendarle pa je to še vedno isti pomnilnik kot pri zaporednem programiranju. Pri programiranju na grafičnih procesnih enotah je to seveda drugače. Pomnilnik grafične procesne enote je ločen in do njega ne moremo dostopati na enak način kot do glavnega pomnilnika računalnika. Prav tako moramo razmisliti o prenosih podatkov iz glavnega pomnilnika računalnika v pomnilnik grafične procesne enote in obratno, saj ti trajajo kar nekaj časa in jih poskušamo čim bolj zmanjšati. Bolj kompleksna je tudi delitev dela, ki lahko ključno vpliva na hitrost izračuna. Za učinkovito delitev dela pa moramo bolje poznati splošno arhitekturo grafičnih procesnih enot, ki je opisana v nadaljevanju.

2.1 Arhitektura grafične procesne enote

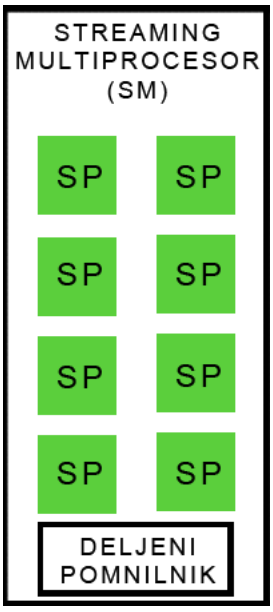
Ne glede na tip imajo vse grafične procesne enote podobno zasnovo, ki se zelo razlikuje od centralne procesne enote. Vzrok za to lahko najdemo v naravi primarne naloge grafičnih procesnih enot - izrisu slike. Izris slike je proces, ki zahteva relativno veliko računskih operacij, ki pa navadno niso zelo zahtevne.

Grafična procesna enota namreč na zaslon izrisuje primitive, ki jim je pri vsakem izrisu potrebno izračunati določene atribute. Točno število primitivov na posamezni sliki je odvisno od posamezne scene, ki jo želimo izrisati, ločljivosti zaslona in še nekaterih drugih dejavnikov. Če k velikemu številu primitivov za izris vsake slike dodamo še število izrisov na sekundo, ki mora biti za tekočo animacijo večje vsaj od števila 25, dobimo zavidljivo številko.

Zaradi zgoraj omenjenih razlogov je idealno, da ima grafična procesna enota veliko število aritmetično logičnih enot, ki skrbijo za hiter izračun. Tako lahko proces izrisa

poteka vzporedno v nekaj sto ali nekaj tisoč jedrih, kar odločno skrajša čas, potreben za izris slike na zaslon. Kontrolna enota grafične procesne enote je nekoliko manj zmogljiva, to pa ne predstavlja večje omejitve, saj v kodi, ki se izvaja na grafični procesni enoti, navadno ni veliko vejitev.
Jedra so navadno organizirana v skupine ali bloke, ki vsebujejo več jeder. Samo poimenovanje se med posameznimi proizvajalci nekoliko razlikuje. Proizvajalec *nVidia* take skupine jeder imenuje *Streaming Multiprocessor*. Vsaka skupina vsebuje več primitivnih jeder, ki jih pri proizvajalcu *nVidia* imenujejo *Streaming Processor*. Vsa jedra znotraj enega SM si delijo lokalni pomnilnik. Grafično je to prikazano na sliki 2.1. Na voljo jim je še globalni pomnilnik, pomnilnik konstant in pomnilnik tekstur, ki so skupni vsem jedrom.

Drugi pomembni del arhitekture grafične procesne enote je torej pomnilnik. Pogosto se zgodi, da so ozko grlo pri izračunih ravno vodila od procesnega dela do pomnilniškega dela grafične procesne enote. Kot je navedeno zgoraj, ima grafična procesna enota navadno več različnih pomnilnikov, ki se med seboj razlikujejo po načinu in hitrosti dostopa ter velikosti. Deljeni pomnilnik, ki je skupen le jedrom znotraj enega bloka jeder, je zelo hiter, a je manjši. Dodatna omejitev je, da do njega lahko dostopajo le tista jedra, ki so v istem bloku. Globalni pomnilnik je veliko večji, do njega lahko dostopajo vsa jedra, dostop pa je veliko počasnejši. Pomnilnika konstant in tekstur sta prav tako skupna vsem jedrom in sta manjša od globalnega pomnilnika. V pomnilnik konstant se shranijo konstante in argumenti za izvajanje ščepcev, pomnilnik tekstur pa je optimiziran za hranjenje tekstur oziroma slik.



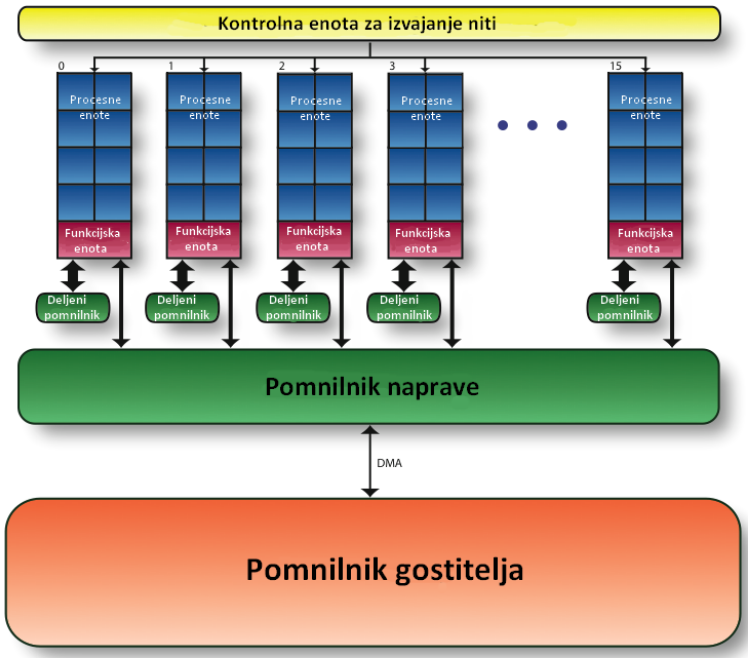
Slika 2.1: CUDA SM

Pri programiranju na grafični procesni enoti je najpomembneje, da se zavedamo, koliko časa je potrebnega za prenos podatkov iz glavnega pomnilnika računalnika v pomnilnik grafične procesne enote. Točne številke

so odvisne od vodila. Na novejših osnovnih ploščah je temu namenjeno vodilo *PCI Express* ali krajše *PCIe*. Trenutno je v uporabi vodilo *PCIe 2.0 x16*, ki omogoča hitrosti do 8 GB na sekundo. Kljub temu, da se to sliši kar veliko, je v primerjavi s prenosi med procesnim in pomnilniškim delom grafične procesne enote zelo malo. Tukaj so hitrosti reda nekaj sto GB na sekundo. Nekaj prenosom se v določenih primerih seveda ne moremo izogniti. Če potrebujemo za začetek računanja na grafični procesni enoti začetne podatke, jih je potrebno pred prvim izvajanjem ščepca pač prenesti v pomnilnik grafične procesne enote. Včasih po končanem izračunu potrebujemo podatke v glavnem pomnilniku računalnika in zopet potrebujemo prenos podatkov, pomembno pa je, da čim bolj zmanjšamo število takih prenosov. Lep primer je izračun podatkov z uporabo grafične procesne enote in zatem grafični izris teh podatkov. V tem primeru je potrebno sam program zasnovati tako, da se med izračunom in izrisom podatki ne prenašajo v glavni pomnilnik računalnika. S takim primerom se srečamo tudi pri izračunu valovne enačbe in vizualizaciji površine po vsakem koraku izračuna.
V določenih situacijah lahko nekaj časa prihranimo tudi z uporabo deljenega pomnilnika ali pomnilnikov konstant in tekstur, vendar pa to ni vedno mogoče in je odvisno od narave problema, ki ga rešujemo na grafični procesni enoti.
Celotna arhitektura sodobne GPE in povezave z glavnim pomnilnikom računalnika je v splošnem predstavljena na sliki 2.2.

2.2 Omejitve paralelizma

Glavni cilj paralelnega programiranja je pohitritev izvajanja algoritma. Določen problem želimo rešiti čim hitreje in če nam serijska različica algoritma ne nudi zadovoljive hitrosti, je pogosto rešitev v paralelizmu. Pri nekaterih problemih lahko tako dosežemo zavidljive pohitritve, vse pa ima neko mejo.
Pred pričetkom programiranja je vedno smiselno oceniti teoretične meje pohitritev in se na podlagi tega odločiti o smiselnosti paralelnega programiranja.



Slika 2.2: Arhitektura sodobne GPE,
povzeto po http://www.hpcwire.com/

2.2.1 Amdahlov zakon

Vsak algoritem je sestavljen iz deleža programske kode, ki se lahko izvaja paralelno in deleža, ki se mora izvesti zaporedno. Čas, ki ga za izvajanje potrebuje prvi delež, lahko z ustreznim programiranjem in ustrezno strojno opremo zelo pohitrimo, tudi tukaj pa obstajajo meje. Večjo težavo navadno predstavlja tisti del programske kode, ki se mora izvesti zaporedno. Tukaj s paralelizmom ne dosežemo prav nič. V tem primeru je edini način za pohitritev izvajanja zmogljivejša strojna oprema.
To zelo dobro opisuje *Amdahlov zakon*, ki določa teoretično pohitritev algoritma, če ga izvaja p niti. Poznati pa je potrebno delež algoritma, ki ga je mogoče izvajati paralelno in delež, ki se mora izvajati zaporedno. Čas, ki ga algoritem potrebuje za izvajanje na n nitih je določen z enačbo $T(p) = T(1)(f + \frac{1}{p}(1 - f))$, kjer f določa delež algoritma, ki ga ni mogoče izvajati paralelno. Pohitritev je torej definirana z

enačbo:

$$S(p) = \frac{T(1)}{T(p)} = \frac{1}{f + \frac{1}{p}(1-f)} \quad .$$

Zavedati pa se moramo, da so to teoretične meje, ki jih je v realnem svetu nemogoče doseči. Taki teoretični izračuni predpostavljajo idealne pogoje in ne upoštevajo časa, potrebnega za prenose podatkov, komunikacijo in podobno.
Primer uporabe Amdahlovega zakona na večjedrnih sistemih si bralec lahko ogleda v [3].
Po Amdahlovem zakonu vidimo, da zelo majhen procent programske kode, ki se mora izvesti zaporedno, lahko zelo omeji pohitritve, ki jih lahko dosežemo. Gustafson je odkril protiprimer Amdahlovemu zakonu in osnoval nekoliko spremenjen Gustafsonov zakon [4].

2.2.2 Raztegljivost

Če potrebujemo večje pohitritve, lahko posežemo po boljši strojni opremi, ki omogoča izvajanja več niti naenkrat, ni pa nujno, da bo pohitritev blizu pričakovanjem. Da bi vedeli, kaj lahko pričakujemo ob povečanju števila niti, je potrebno oceniti raztegljivost ali skalabilnost problema. Pohitritev lahko zapišemo kot:

$$S(n,p) \leq \frac{\sigma(n) + \rho(n)}{\sigma(n) + \rho(n)/p + \kappa(n,p)}$$

ali

$$S(n,p) \leq \frac{p(\sigma(n) + \rho(n))}{\sigma(n) + \rho(n) + (p-1)\sigma(n) + p\kappa(n,p)} \quad ,$$

kjer σ predstavlja čas za izvajanje zaporednega dela algoritma, ρ čas za izvajanje paralelnega dela algoritma, κ čas za komunikacijo in p število niti. Vsota časov, ki ga procesi porabijo za dodatno komunikacijo, je $T_{komunikacija} = (p-1)\sigma(n) + p\kappa(n,p)$. Pohitritev torej lahko zapišemo tudi tako:

$$S(n,p) \leq \frac{pT_s(n)}{T_s(n) + T_{komunikacija}(n,p)} \quad .$$

Učinkovitost je definirana z enačbo:

$$E(n,p) = \frac{S(n,p)}{p} \leq \frac{T_s(n)}{T_s(n) + T_{komunikacija}(n,p)} \quad .$$

Če izrazimo sekvenčni čas dobimo:

$$T_s(n) \geq \frac{E(n,p)}{1-E(n,p)} T_{komunikacija}(n,p) = CT_{komunikacija}(n,p) \quad .$$

Želimo, da je C konstanta. $T_{komunikacija}$ narašča s stopnjo paralnosti p. Če povečamo n, lahko držimo razmerje konstantno.

Poglavje 3

Valovna enačba

Valovna enačba, kot razkriva že samo ime, opisuje valovanje. Gre za vse vrste valovanja, ki se pojavljajo v svetu fizike. Kot primer lahko vzamemo valovanje zvoka ali valovanje tekočin. V tej diplomski nalogi sem se osredotočil na valovanje tekočin.

Valovanje lahko predstavimo s parcialno diferencialno enačbo drugega reda. Splošna enačba se glasi:

$$\frac{\partial^2 \vec{u}}{\partial t^2} = c^2 \Delta \vec{u} \quad . \tag{3.1}$$

V enačbi (3.1) $u(x,t)$ predstavlja funkcijo časa t in prostorskih koordinat x, c predstavlja hitrost valovanja, Δ pa Laplaceov operator.

Enačba (3.1) je splošna, v diplomskem delu pa se bom osredotočil na tridimenzionalni prostor.

Ker enačbe v splošnem analitično ni moč rešiti, je problem valovanja potrebno aproksimirati. Aproksimacijo sem izvedel z Eulerjevo metodo [5], ki je računsko manj zahtevna, a tudi zelo nestabilna in metodo Runge Kutta 4. reda [5], ki je veliko bolj stabilna, vendar računsko zahtevnejša.

V realnem svetu nobeno valovanje ne traja neskončno dolgo časa, če nanj ves čas ne deluje neka sila. Če želimo doseči ta učinek, je potrebno upoštevati dušenje valovanja.

3.1 Eulerjeva metoda

Eulerjeva metoda je preprosta numerična aproksimacija, s katero lahko aproksimiramo rešitve navadnih diferencialnih enačb z določenim začetnim stanjem. Navadne diferencialne enačbe so tiste enačbe, v katerih je odvisna spremenljivka funkcija ene neodvisne spremenljivke. Splošna enačba Eulerjeve metode je:

$$y_{n+1} = y_n + f(t_n, y_n) \cdot h \quad , \tag{3.2}$$

kjer h predstavlja iteracijski korak.
Ker je valovna enačba (3.1) navadna diferencialna enačba drugega reda, jo lahko enostavno aproksimiramo z uporabo Eulerjeve metode.
Prednost Eulerjeve metode je predvsem računska nezahtevnost, težavo pa predstavljata relativno velika računska napaka in numerična nestabilnost.
Ker je Eulerjeva metoda metoda prvega reda, ima lokalno napako (napako v vsakem koraku) reda $\mathcal{O}(h^2)$ in globalno napako (napako po določenem času t) reda $\mathcal{O}(h)$.
Poleg velike napake se pojavlja še numerična nestabilnost, kar pomeni, da je lahko absolutna vrednost rezultata pridobljenega s to metodo v primerjavi s točnim rezultatom zelo velika. Numerična nestabilnost je pri večjih korakih h veliko večja.
Numerična nestabilnost Eulerjeve metode je lepo vidna na primeru linearne enačbe

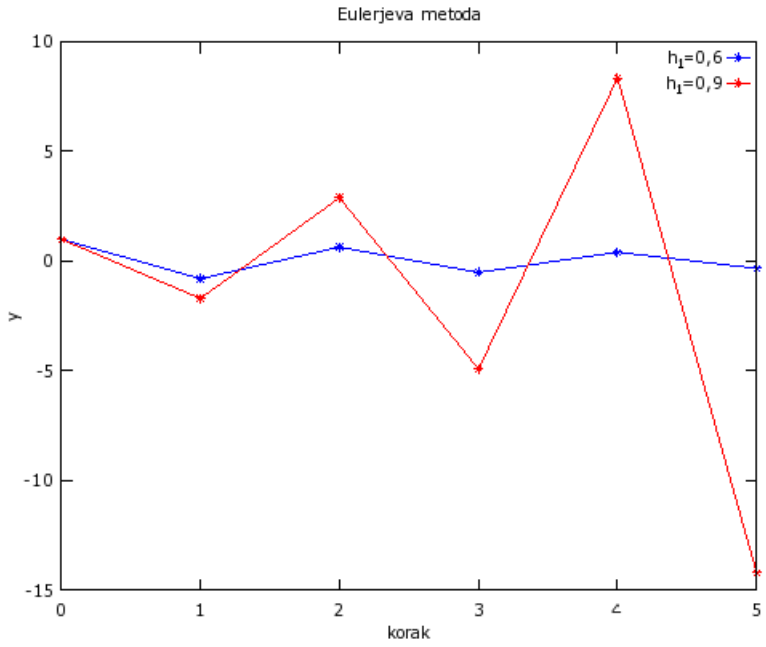
$$\dot{y} = -3y \quad , \quad y(0) = 1 \quad . \tag{3.3}$$

Diskretna različica enačbe 3.3 je:

$$y_{n+1} = y_n - 3y_n \cdot h \quad , \tag{3.4}$$

$$t_{n+1} = t_n + h \quad . \tag{3.5}$$

Natančna rešitev enačbe je $y(t) = e^{-3t}$, kar se za velike t približuje vrednosti 0. Če enačbo rešimo še po Eulerjevi metodi s korakom $h = 0{,}9$, vidimo, da se enačba ne približuje vrednosti 0, pač pa alternirajo vedno večja pozitivna in negativna števila. K sreči pa se to navadno dogaja le pri večjih korakih. V tem primeru se težave znebimo že pri koraku $h = 0{,}6$.



Slika 3.1: Rešitev diferencialne enačbe (3.3) z Eulerjevo metodo

3.2 Metoda Runge-Kutta 4. reda

Metode Runge-Kutta so podobne Eulerjevi metodi, le da so nekoliko naprednejše in računsko zahtevnejše, so pa zato veliko bolj stabilne in natančnejše od Eulerjeve metode. Z njimi prav tako numerično aproksimiramo navadne diferencialne enačbe. Metode se med seboj razlikujejo po redu. Metoda Runge-Kutta 1. reda je pravzaprav enaka Eulerjevi metodi, najbolj uporabljana pa je Runge-Kutta 4. reda. Splošne enačbe so:

$$\dot{y} = f(t, y) \quad , \quad y(t_0) = y_0 \quad , \tag{3.6}$$

$$^1k = f(t_n, y_n) \cdot h \quad , \tag{3.7}$$

$$^2k = f(t_n + \frac{h}{2}, y_n + \frac{1}{2} \cdot^1 k) \cdot h \quad , \tag{3.8}$$

$$^3k = f(t_n + \frac{h}{2}, y_n + \frac{1}{2} \cdot^2 k) \cdot h \quad , \tag{3.9}$$

$$^4k = f(t_n + h, y_n +^3 k) \cdot h \quad , \tag{3.10}$$

$$y_{n+1} = y_n + \frac{1}{6}({}^1k + 2 \cdot {}^2k + 2 \cdot {}^3k + {}^4k) \quad , \tag{3.11}$$

$$t_{n+1} = t_n + h \quad . \tag{3.12}$$

Izračun valovanja površine poteka po podobnem principu kot pri Eulerjevi metodi, le da je pri metodi Runge-Kutta 4. reda več vmesnih korakov. Vmesni rezultati so v zgornjih enačbah predstavljeni s spremenljivkami ${}^1k$, ${}^2k$, ${}^3k$ in ${}^4k$, kjer številke $1, 2, 3$ in 4 predstavljajo indeks.

Ker je to metoda 4. reda, ima lokalno napako (napako na vsak korak) reda $\mathcal{O}(h^5)$, globalno napako (napako po določenem času t) pa reda $\mathcal{O}(h^4)$.

Če diferencialno enačbo 3.3 diskretiziramo za izračun po metodi Runge-Kutta 4. reda dobimo:

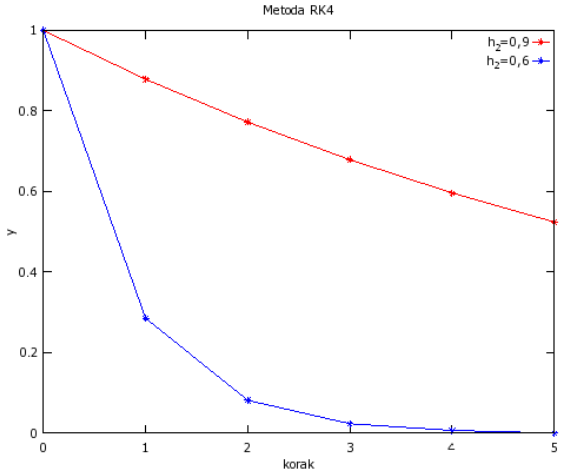
$${}^1k = -3y_n \cdot h \quad , \tag{3.13}$$

$${}^2k = -3(y_n + \frac{1}{2} \cdot {}^1k) \cdot h \quad , \tag{3.14}$$

$${}^3k = -3(y_n + \frac{1}{2} \cdot {}^2k) \cdot h \quad , \tag{3.15}$$

$${}^4k = -3(y_n + {}^3k) \cdot h \quad , \tag{3.16}$$

$$y_{n+1} = y_n + \frac{1}{6}({}^1k + 2 \cdot {}^2k + 2 \cdot {}^3k + {}^4k) \quad . \tag{3.17}$$



Slika 3.2: Rešitev diferencialne enačbe (3.3) z metodo RK4

Izračun diferencialne enačbe z metodo Runge-Kutta 4. reda s korakom $h = 0{,}9$ in $h = 0{,}6$ je prikazan na sliki 3.2. Vidimo lahko, da je izračun po metodi Runge-Kutta 4. reda stabilen že pri $h = 0{,}9$.

3.3 Izvedba

Ker je v računalništvu vse diskretno, je potrebno tudi površino, na kateri računamo valovanje, diskretizirati. Najprej je potrebno določiti širino in dolžino površine. Površino nato predstavimo z dvema poljema točk. Vsako polje vsebuje toliko točk, kolikor znaša zmnožek širine in dolžine površine. Prvo polje, ki ga lahko poimenujemo U, predstavlja višino površine v posamezni točki. Drugo polje, ki ga lahko poimenujemo V, pa njihove hitrosti.
Za izračun je potrebno osnovno enačbo valovanja nekoliko poenostaviti in prilagoditi zgoraj omenjeni predstavitvi površine. Drugi odvod v enačbi (3.1) predstavlja pospešek točke. Za osnovo pri aproksimaciji pospeška uporabimo seštevek razlik višine gladin med sosednjimi točkami. V vsakem računskem koraku tako potrebujemo vrednosti sosednjih točk. Vsak korak je torej podoben konvoluciji.
Da enačbo 3.1 lažje prevedemo v diskretno obliko, jo prevedemo na enačbi:

$$\dot{u} = v \quad , \tag{3.18}$$

$$\dot{v} = c^2 \Delta u \quad . \tag{3.19}$$

Vsako izmed enačb 3.18 in 3.19 posebej diskretiziramo. Zato uvedemo časovni korak h in prostorski korak d.

$$t_{n,+1} = t_n + h \tag{3.20}$$

$$U_{i+1,j} = U_{i,j} + d \quad , \tag{3.21}$$

$$U_{i,j+1} = U_{i,j} + d \quad . \tag{3.22}$$

Če enačbi 3.18 in 3.19 prevedemo v obliko, primerno za numerični izračun po Eulerjevi metodi in upoštevamo časovno in prostorsko diskretizacijo, ki je predstavljena z enačbami 3.20, 3.21 in 3.22, dobimo:

$$U_{i,j,n+1} = U_{i,j,n} + V_{i,j,n+1} \cdot h \quad , \tag{3.23}$$

$$V_{i,j,n+1} = V_{i,j,n} + f(t_n, U_{i,j,n}) \cdot h \quad , \tag{3.24}$$

kjer je

$$f(t_n, U_{i,j,n}) = (U_{i+1,j,n} + U_{i-1,j,n} + U_{i,j+1,n} + U_{i,j-1,n} - 4U_{i,j,n}) \cdot \frac{c^2}{d^2} \quad . \tag{3.25}$$

V enačbi 3.25 je uporabljena konvolucija, ki aproksimira delovanje Laplaceovega operatorja. Indeks n označuje zaporedno številko koraka.
Dušenje valovanja v tem sistemu enačb ni upoštevano. Če želimo sistemu dodati še dušenje, je potrebno v vsakem koraku pri izračunu hitrosti odšteti nek delež vrednosti hitrosti v prejšnjem koraku. Delež lahko predstavimo kot koeficient dušenja k.

$$V_{i,j,n+1} = V_{i,j,n} \cdot (1 - k) + f(t_n, U_{i,j,n}) \cdot h \quad . \tag{3.26}$$

Za aproksimacijo z metodo Runge-Kutta 4. reda je potrebno enačbe za izračun hitrosti prilagoditi. Pri metodi Runge-Kutta 4. reda imamo več vmesnih izračunov za vsako prostorsko točko, ki jih shranjujemo v matrike. Matrike so velikosti zmnožka širine in dolžine površine ter jih lahko poimenujemo ${}^1K$,${}^2K$,${}^3K$ in ${}^4K$. Dobimo:

$${}^1K_{i,j} = f(t_n, U_{i,j,n}) \cdot h \quad , \tag{3.27}$$

$${}^2K_{i,j} = f(t_n + \frac{h}{2}, U_{i,j,n} + \frac{1}{2} \cdot {}^1K_{i,j}) \cdot h \quad , \tag{3.28}$$

$${}^3K_{i,j} = f(t_n + \frac{h}{2}, U_{i,j,n} + \frac{1}{2} \cdot {}^2K_{i,j}) \cdot h \quad , \tag{3.29}$$

$${}^4K_{i,j} = f(t_n + h, U_{i,j,n} + {}^3K_{i,j}) \cdot h \quad , \tag{3.30}$$

$$V_{i,j,n+1} = V_{i,j,n} + \frac{1}{6} \cdot ({}^1K_{i,j} + 2 \cdot {}^2K_{i,j} + 2 \cdot {}^3K_{i,j} + {}^4K_{i,j}) \quad . \tag{3.31}$$

Dušenje valovanja pri izračunu z metodo Runge-Kutta 4. reda vpeljemo na enak način kot pri Eulerjevi metodi, kar prikazuje enačba 3.26.

Pred začetkom reševanja sistema enačb je potrebno določiti še začetne in robne pogoje. Začetni pogoji so določeni z naključno generirano površino, pri robnih pogojih pa imamo dve možnosti. Točke na robovih so lahko vpete, kar pomeni, da se njihova višina ne spreminja, ali proste, kar pomeni, da valujejo na podoben način kot ostale točke na površini.

V prvem primeru je potrebno iz operacije konvolucije izvzeti točke na robovih, saj je tam višina ves čas enaka. V drugem primeru pa je potrebno nekoliko spremeniti konvolucijsko matriko, saj moramo upoštevati le vrednosti točk s pozitivnimi indeksi.

V diplomski nalogi sem se odločil za drugo možnost. V praktičnem smislu to pomeni, da je potrebno pri vsakem koraku konvolucije preveriti, če so sosednji indeksi še del polj U in V.

Poglavje 4

nVidia CUDA

Compute Unified Device Architecture ali krajše CUDA [6] je platforma za programiranje na grafični procesni enoti. Programerjem omogoča uporabo grafične procesne enote za namene računanja. CUDA prinaša širok nabor ukazov, ki omogočajo izvajanje akcij na grafični procesn enoti in dostop do pomnilnika grafične procesne enote. Širši javnosti je bila predstavljena konec leta 2007 skupaj s serijo grafičnih procesnih enot *G80*, ki je bila prva serija, ki je podpirala *nVidia CUDA*. Prva grafična procesna enota z arhitekturo CUDA je bila *nVidia GeForce 8800 GTX*.
CUDA je na voljo kot razširitev programskih jezikov *C*, *C++* in *Fortran*. Tudi z nekaterimi drugimi programskimi jeziki je mogoče uporabljati prednosti, ki jih prinaša CUDA. Drugi programski jeziki sicer nimajo uradne podpore, obstajajo pa nekatere knjižnice, ki omogočajo uporabo funkcionalnosti CUDA.
Platforma CUDA pa ima eno veliko omejitev, ki sicer ni vezana na programsko pač pa na strojno opremo. Omogoča poganjanje programov le na grafičnih procesnih enotah proizvajalca *nVidia*, saj so le te grafične procsne enote združljive s CUDA arhitekturo.
Grafične procesne enote pred arhitekturo CUDA so imele dva ločena tipa procesnih enot. Prvi so senčilniki slikovnih točk *(angl. pixel shaders)*, drugi pa senčilniki vozlišč *(angl. vertex shaders)*. Pojavile so se težave v neizkoriščenosti ene ali druge skupine pri določenih opravilih. To težavo arhitektura CUDA v celoti odpravi, saj so si aritmetično logične enote med seboj enakovredne. Hkrati pa je to idealno za

uporabo grafične procesne enote za računske operacije.

4.1 Arhitektura

Grafična procesna enota je sestavljena iz večjega števila aritmetično logičnih enot. Pri arhitekturi CUDA so ALE na čipu med seboj enakovredne in lahko opravljajo iste operacije. Tako lahko katerakoli ALE prevzame katerokoli nalogo, ki čaka v vrsti za izvedbo. Enote ALE so v arhitekturi CUDA razširjene tako, da podpirajo IEEE 754 standard [7], ki določa računanje v plavajoči vejici. Podrobneje je predstavljen v [8]. Razširjen je tudi nabor ukazov, ki omogoča več splošno namenskih operacij.
Posamezne izvajalne enote, ki jih *nVidia* imenuje enota SP *(angl. Streming Processor)*, so združene v skupine jeder, ki se imenujejo enote SM *(angl. Streaming Multiprocessor)* [9].
Vsakemu *SM* pripada deljeni pomnilnik *(angl. Shared Memory)*, ki je veliko hitrejši od globalnega pomnilnika. Ne smemo ga zamenjati z lokalnim pomnilnikom *(angl. Local Memory)*, ki je pravzaprav del globalnega pomnilnika, vanj pa vsaka nit shranjuje podatke, kot so na primer začasne spremenljivke. Do deljenega pomnilnika lahko dostopajo le *SP* znotraj ene *SM* in ga ne moremo uporabljati na isti način kot globalni pomnilnik. Ena *SM* torej predstavlja nekakšno zaključeno celoto. V izvajanje dobi vsaka svoj blok niti. Posamezne niti znotraj tega bloka razdeli med svoje *SP*. Za izmenjavo informacij med posameznimi *SM* je na voljo globalni pomnilnik, ki pa je veliko počasnejši od deljenega. Vseeno pa zaradi velikosti globalnega pomnilnika in narave problemov večino podatkov shranjujemo prav v globalni pomnilnik. Vsaka grafična procesna enota ima še pomnilnik konstant in tekstur, ki sta zelo hitra, a relativno majhna ter bolj specifična. Najhitrejši pomnilnik predstavljajo registri, ki pa so zelo majhni. Ko zmanjka registrov, se podatki začnejo shranjevati v lokalni pomnilnik, kar lahko drastično upočasni izvajanje programa. Vse vrste pomnilnikov, dostopni časi in omejitve dostopa so zbrani v tabeli 4.1.
Kot vidimo, se posamezni pomnilniki po hitrosti dostopa med seboj zelo razlikujejo. Zato nam pametno shranjevanje podatkov v določenih primerih lahko pohitri izvajanje algoritma. Vedeti pa moramo, da ima globalni pomnilnik predpomnilnik, ki mu pomaga skriti visoke dostopne čase. To pride še posebej do izraza pri no-

pomnilnik	kdo lahko dostopa	čas dostopa
registri	posamezna nit	1 urina perioda
deljeni pomnilnik	niti znotraj bloka (SM)	5 urinih period
lokalni pomnilnik	posamezna nit	500 urinih period
globalni pomnilnik	vsi	500 urinih period
pomnilnik konstant	vsi	5 urinih period
pomnilnik tekstur	vsi	5 urinih period

Tabela 4.1: Pomnilniška hierarhija v arhitekturi CUDA

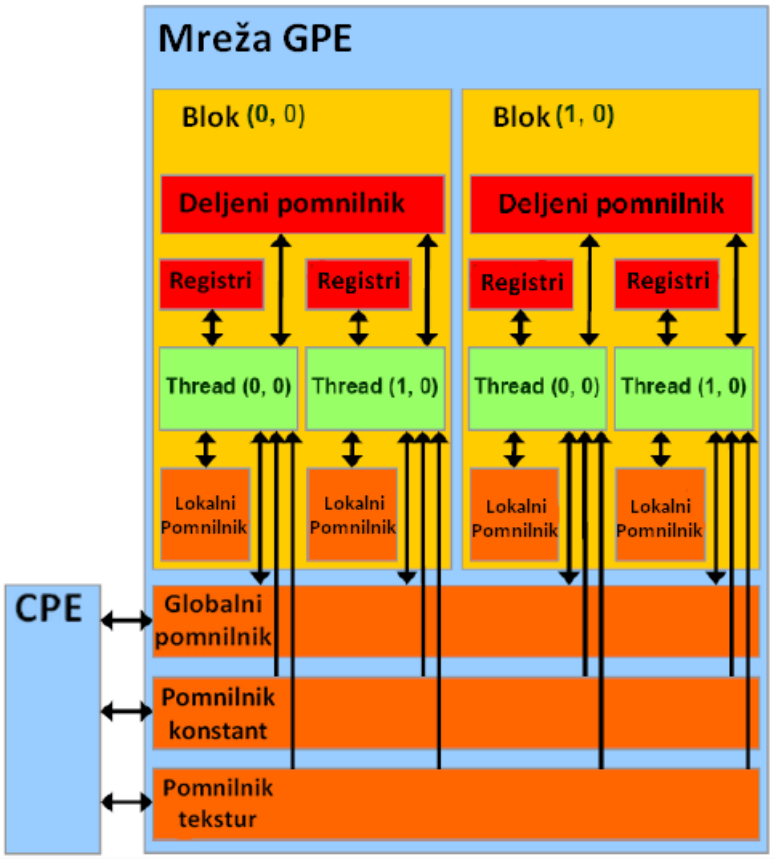
vejših grafičnih procesnih enotah, ki imajo vedno večji predpomnilnik in je branje iz globalnega pomnilnika pravzaprav relativno redek pojav. Pomnilniška hierarhija je grafično prikazana na sliki 4.1.

4.2 Izvajalni model

Glavna prednost programiranja na grafičnih procesnih enotah je veliko število niti, ki se izvajajo paralelno na ločenih ALE. Tudi pri arhitekturi CUDA je to osrednja ideja. Uporabnik določi število niti, ki je lahko izjemno veliko. Niti se na grafični procesni enoti dinamično razvrščajo in izvajajo paralelno.

Program, napisan za arhitekturo CUDA, lahko razdelimo na več enot. Vsak program vsebuje delež serijske kode, ki se ne izvaja na grafični procesni enoti, ampak na gostitelju, ki je običajno CPE. S serijsko kodo navadno poskrbimo za inicializacijo in čiščenje po koncu izvajanja paralelne kode. V inicializacijo spada tudi prenos paralelne kode na grafično procesno enoto in prenos podatkov iz pomnilnika gostitelja v pomnilnik grafične procesne enote. Tisti del kode, ki se paralelno izvaja na GPE, imenujemo *ščepec* (angl. kernel).

Ščepec izvaja več niti naenkrat, ki opravljajo iste operacije nad različnimi podatki. Niti so logično porazdeljene v polje. Več niti skupaj sestavlja blok. Blok v izvajanje dobi posamezen *SM*, med seboj pa te niti lahko komunicirajo preko deljenega pomnilnika. Med seboj neodvisni bloki niti sestavjajo mrežo niti *angl. grid* in med



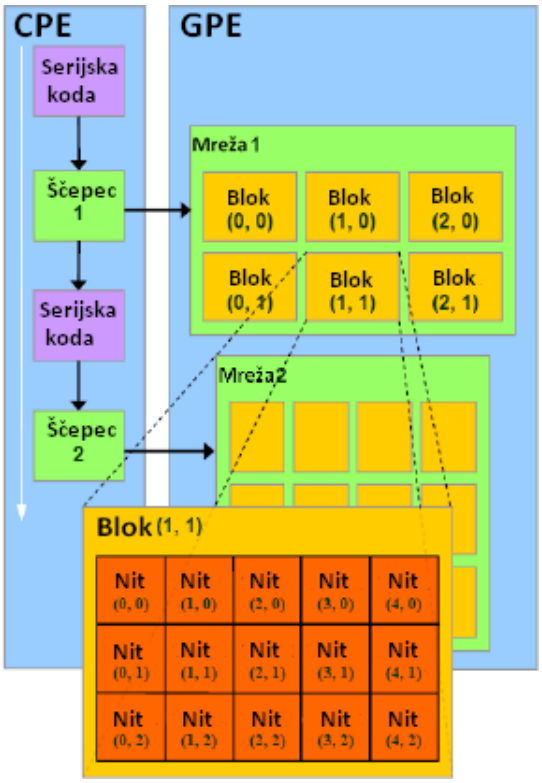
Slika 4.1: Pomnilniški model CUDA,
povzeto po http://3dgep.com/

seboj komunicirajo preko globalnega pomnilnika. Posamezni bloki se lahko izvajajo v poljubnem vrstnem redu, ki ga programer ne pozna. Bloki v mreži so logično razporejeni v 2D polje, vsak blok je torej viden preko 2D indeksov. Niti v blokih pa so logično razporejene v 2D ali 3D polje, vsaka nit je torej vidna preko 2D ali 3D indeksov. Tako indeksiranje nam omogoča, da izvajanje operacij nad podatki v ščepcu prilagodimo glede na posamezno nit. Grafično je organizacija niti in blokov prikazana na sliki 4.2.

Paralelni model arhitekture CUDA je bolj natančno predstavljen v delu [10] in [11], kjer si bralec lahko ogleda tudi izsledke ob implementaciji rešitev različnih proble-

mov.

Kljub dinamičnemu razvrščanju in visoki stopnji paralelizma obstajajo nekatere



Slika 4.2: Organizacija niti,
povzeto po http://ixbtlabs.com/

omejitve. Novi ščepec ne more začeti z izvajanjem, dokler se ne izvedejo vse niti prejšnjega ščepca. Grafična procesna enota izvaja le en ščepec naenkrat. Programer ima možnost sinhronizacije oziroma postavitve prepreke. Znotraj ščepca lahko sinhronizira le niti, ki pripadajo istemu bloku, izven ščepca pa lahko sinhronizira vse niti.

Platforma CUDA zaradi svoje relativne enostavnosti in majhne količine dodatne kode, ki jo mora programer obvladati za samo inicializacijo, predstavlja dobro izbiro za učenje programiranja na grafični procesni enoti. Prinaša pa žal omejitve z vidika strojne opreme.

Poglavje 5

OpenCL

Open Computing Language ali krajše OpenCL [12] je, tako kot CUDA, platforma za paralelno programiranje. Za razliko od CUDA pa OpenCL ni vezan le na grafične procesne enote. Omogoča izvajanje na različnih platformah, med drugim tudi na centralnih procesnih enotah, FPGA vezjih in ostalih procesnih enotah. Zasnovan je bil kot programski jezik, ki bo omogočal izvajanje na heterogenem sistemu. OpenCL prinaša ukaze, ki bazirajo na programskem jeziku C in programerju omogočajo programiranje na prej omenjenih napravah.

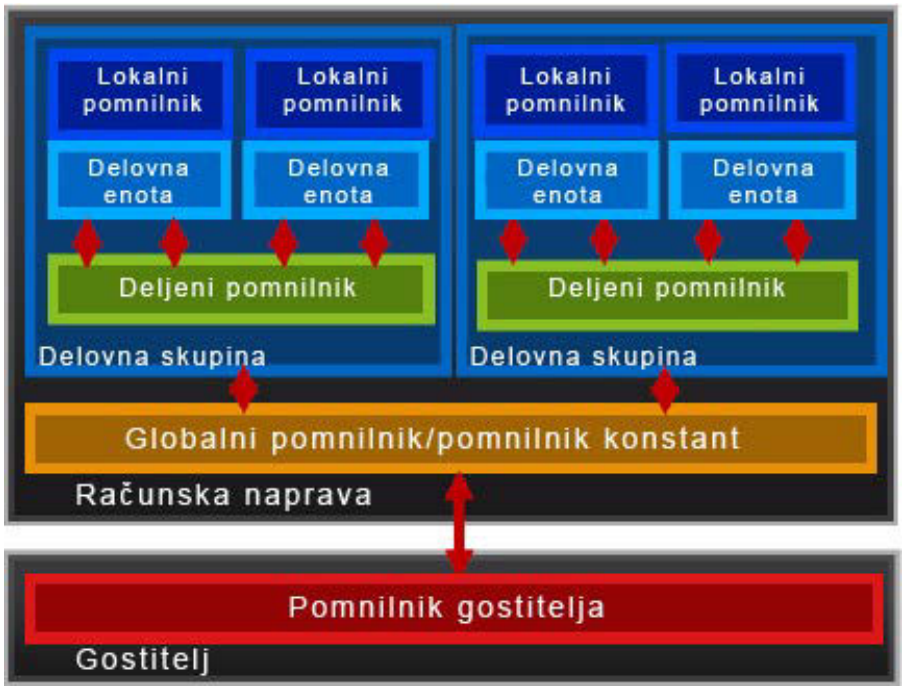
OpenCL je odprt standard, ki spada pod okrilje neprofitne organizacije *Khronos Group*, ki danes združuje vse večje proizvajalce strojne opreme. Razvoju so se kmalu pridružila skoraj vsa velika podjetja s področja računalništva, kar je standardu omogočilo hitro širitev in podporo različni strojni opremi večih proizvajalcev. OpenCL se pogosto uporablja za programiranje na grafičnih procesnih enotah, saj uporabniku med drugim nudi tudi ukaze za upravljanje s pomnilnikom GPE in poganjanje kosov programske kode (ščepcev) na GPE.

OpenCL 1.0 je bil javnosti najprej predstavljen v okviru operacijskega sistema *Mac OS X Snow Leopard*. Kmalu zatem sta dve največji podjetji, ki se ukvarjata s proizvodnjo grafičnih procesnih enot, začeli nuditi podporo za platformo OpenCL. To je bila velika prelomnica za standard in krivulja razvoja je začela strmo rasti. V času pisanja je najnovejša različica OpenCL 2.0, ki je bila predstavljena konec leta 2013. OpenCL pridobiva vedno več podpornikov, verjetno tudi zaradi široke podpore naj-

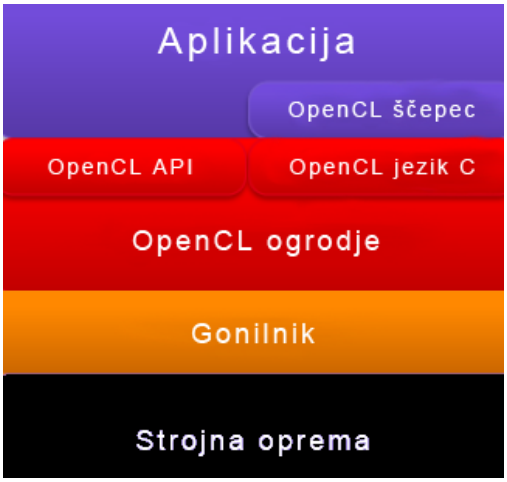
različnejših proizvajalcev strojne opreme. To standardu zagoravlja visoko stopnjo neodvisnosti in prenosljivosti, kar za konkurenčno platformo CUDA težko trdimo. Prenosljivost v smislu delovanja programa na grafičnih in centralnih procesnih enotah različnih proizvajalcev je pri OpenCL zagotovljena, ni pa nobenega zagotovila glede zmogljivosti. Lahko se zgodi, da bo identičen program na podobno zmogljivi strojni opremi deloval različno hitro. V takih primerih mora programer poskrbeti za različne optimizacije. To še posebej velja, če želimo isti program poganjati tako na grafičnih kot tudi centralnih procesnih enotah. Možne optimizacije so na primer optimalen dostop do pomnilnika, prilagajanje delitve dela, izogibanje uporabi deljenega pomnilnika na CPE in tako naprej [13], [14], [15].

5.1 Arhitektura

O neki splošni arhitekturi strojne opreme pri platformi OpenCL težko govorimo, saj ni omejena le na grafične procesne enote. Vendarle pa gre za zelo podoben princip. Imamo veliko stopnjo paralelizma, kar pomeni, da ustvarimo veliko število niti, ki se dinamično razvrščajo in izvajajo na procesnih enotah.
Med OpenCL in CUDA lahko vidimo marsikatere podobnosti. Zato večino napisanega o arhitekturi in pomnilniški hierarhiji v poglavju 4 velja tudi za OpenCL. Pogosto so razlike le v poimenovanju. OpenCL uporablja termin delovna enota*(angl. workitem)*, ki predstavlja določeno nit in termin delovna skupina *(angl. workgroup)*, ki predstavlja blok niti. Tako kot pri CUDA, ima vsaka nit svoj privatni pomnilnik (ekvivalent lokalnemu pomnilniku pri CUDA), posamezne niti v istem bloku *(angl. workgroup)* pa si delijo lokalni pomnilnik (ekvivalent deljenemu pomnilniku pri CUDA). Tudi tukaj se srečamo z globalnim pomnilnikom, ki omogoča komunikacijo med vsemi nitmi in pomnilnikom konstant. Grafično to prikazuje slika 5.1.
Ker OpenCL podpira različne platforme in različne naprave, ima programer navadno nekoliko več dela pri sami inicializaciji. Preveriti mora prisotne platforme in pripadajoče naprave ter izbrati najprimernejšo. Na podlagi tega OpenCL potem s pomočjo vmesnih členov pripravi napravo, da lahko začne z izvajanjem ščepca. Arhitektura OpenCL z glavnimi elementi je prikazana na sliki 5.2



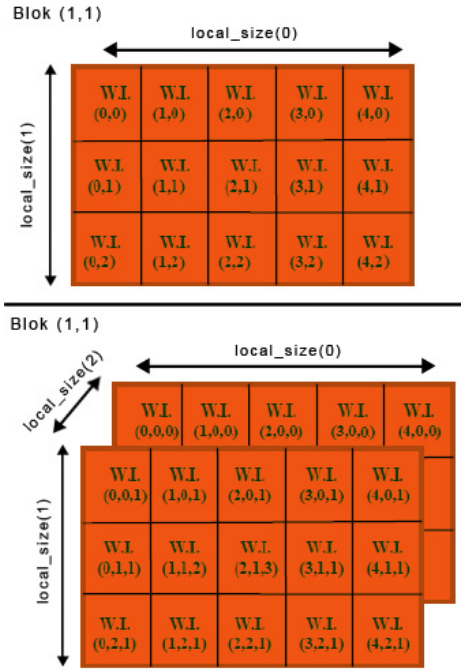
Slika 5.1: Pomnilniški model OpenCL,
povzeto po http://www.codeproject.com/



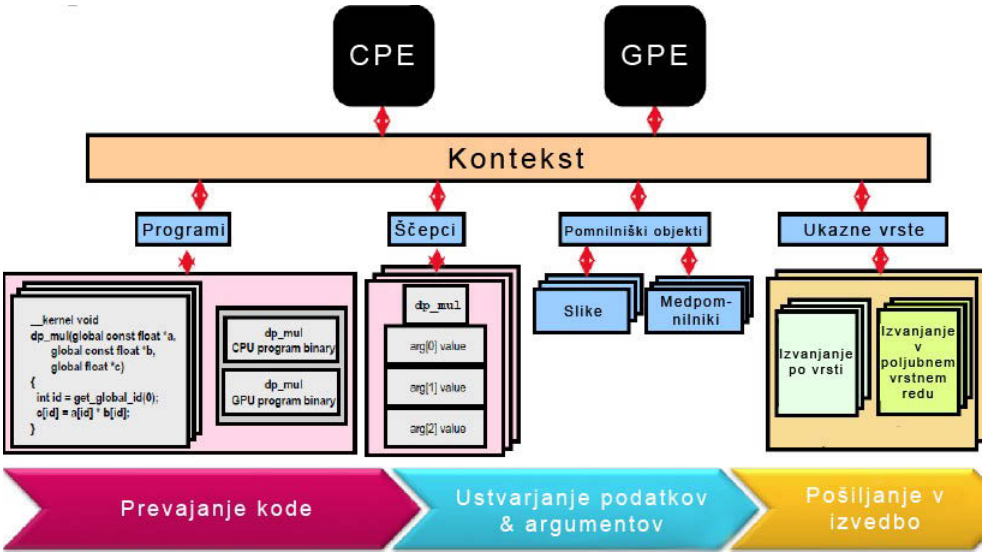
Slika 5.2: Arhitektura platforme OpenCL,
povzeto po http://www.mat.ucsb.edu/

5.2 Izvajalni model

Ker je OpenCL veliko bolj razširljiv in prenosljiv, pri inicializaciji zahteva nekoliko več sodelovanja programerja. Programer je najprej primoran izbrati platformo in pripadajočo napravo. Navadno imamo enega gostitelja in eno ali več računskih naprav. Delo programerja je, da izbere najprimernejšo. Zatem sledi kreiranje konteksta. Kontekst je nekakšno virtualno okolje za izvajanje ščepca, upravljanje s pomnilnikom in sinhronizacijo. Sam princip je tukaj nekoliko bolj zapleten kot pri platformi CUDA. Kontekst združuje množico naprav, pomnilnik in ukazne vrste, kamor lahko programer postavlja ukaze, ki nato izvajajo želene operacije.
Prav vsi OpenCL ukazi so torej podani preko ukaznih vrst. Vsaka naprava mora imeti vsaj eno ukazno vrsto, ki je namenjena le njej. Vsaki ukazni vrsti lahko določimo tip sinhronizacije. Odločamo se med izvajanjem po vrsti *(angl. in-order)* in poljubnim vrstnim redom *(angl. out-of-order)*.
Pred pričetkom izvajanja ščepca je potrebno definirati problemsko področje (velikost problema). Za vsako točko tega področja, ki je lahko predstavljeno kot 1D, 2D ali 3D polje, izvedemo ščepec. Tako kot pri CUDA ščepec izvaja isti program nad različnimi podatki. Primer različne predstavitve problemskega področja prikazuje slika 5.3.
Da se lahko ščepec izvede, ga je potrebno prevesti. To naredimo tako, da ustvarimo programski objekt, ki vsebuje kontekst, programsko kodo, seznam ciljnih naprav in navodila prevajalniku. Ko je programski objekt ustvarjen, mu lahko nastavimo ustrezne argumente. Prevajanje se pri OpenCL zgodi *Just In Time*, kar pomeni, da se program prevede šele tik pred začetkom prvega izvajanja.
Shemo celotnega izvajalnega modela si za boljšo predstavo lahko ogledate na sliki 5.4.
Sintaksa in princip razvoja programov z OpenCL v tem delu nista predstavljena. Bralec lahko to najde v delu [16].



Slika 5.3: 2D in 3D problemsko področje



Slika 5.4: Shema izvajalnega modela OpenCL,
povzeto po http://mohamedfahmed.wordpress.com/

Tudi pri OpenCL obstajajo podobne omejitve izvajanja kot pri CUDA. Nov ščepec mora pred začetkom izvajanja počakati, da z izvajanjem konča ščepec pred njim, programeru pa tudi OpenCL nudi ukaze za sinhronizacijo. Znotraj ščepca so programerju na voljo prepreke, ki sinhronizirajo niti znotraj istega bloka *(angl. workgroup)*, zunaj ščepca pa lahko programer počaka, da z izvajanjem končajo vse niti. Posameznim ukazom, ki jih programer pošilja v ukazne vrste, lahko določi tudi dogodkovni objekt in kasneje v kodi počaka, da se ukaz, ki mu je bil podan nek dogodkovni objekt, izvede.

OpenCL je torej zelo vsestranska platforma za paralelno programiranje, ki med drugim omogoča tudi programiranje na grafičnih procesnih enotah. Zaradi svoje prenosljivosti in prilagodljivosti od programerja zahteva nekaj več programiranja in poznavanja samega izvajalnega modela. Nagrada pa je kar primerna, saj je program, ki uporablja OpenCL, popolnoma strojno neodvisen in prenosljiv.

Poglavje 6

Vizualizacija

V prejšnjih poglavjih so bile predstavljene tehnologije, ki smo jih uporabili za izračun valovne diferencialne enačbe. Ker je rezultat računanja mogoče grafično prikazati, smo v okviru diplomskega dela poskrbeli tudi za vizualizacijo površine po vsakem računskem koraku. Vizualizacijo smo vključili tudi v časovne meritve, saj nas je zanimalo, koliko le ta vpliva na celoten čas, ki se porabi za izračun in prikaz enega koraka. Rezultati so predstavljeni in interpretirani v poglavju 7.

Vizualizacijo sem uporabil v navezi s platformo OpenCL, saj le ta nudi nekoliko več opcij za deljenje grafičnega pomnilnika s standardi za izrisovanje slike. To programerju nekoliko olajša delo, predvsem pa pospeši proces izrisa slike. Prav tako vizualizacija porabi kar nekaj časa in razlike med platformama CUDA in OpenCL v času izračuna v tem primeru niso bistvene.

Da smo lahko v obstoječo kodo, ki je skrbela za izračun diferencialne enačbe, dodali vizualizacijo, je bilo potrebno opraviti kar nekaj sprememb. Površina je bila do sedaj predstavljena s poljem velikosti zmnožka širine in višine. Ker za vizualizacijo za vsako točko potrebujemo tri koordinate, je potrebno površino sedaj predstaviti s poljem trikratne velikosti zmnožka širine in višine. Pred pričetkom računanja je potrebno polje pripraviti. Določiti je potrebno dve koordinati, ki se ne spreminjata. Zaradi tega je bilo potrebno nekoliko spremeniti tudi sam ščepec, ki skrbi za računanje. Poleg tega je časovni korak sedaj variabilen in je odvisen od števila izrisanih slik na sekundo, saj se pred vsakim izrisom zgodi izračun enega koraka.

Za sam izris sem izbral standard OpenGL, ki je predstavljen v nadaljevanju. V navezi z OpenGL sem uporabil še knjižnici FreeGLUT in GLEW, ki poskrbita predvsem za interakcijo z uporabnikom.

6.1 OpenGL

Open Graphics Library ali krajše OpenGL je najpogosteje uporabljan API za ustvarjanje 2D ali 3D interaktivnih grafičnih aplikacij. Predstavljen je bil leta 1992, od takrat pa se je zelo razvil in prehitel konkurente. Še pred nekaj leti je bil standard *Microsoft DirectX* dostojen konkurent standardu OpenGL, v zadnjih letih pa je OpenGL zaradi silovitega razvoja postal najbolj priljubljena izbira za tovrstno programiranje. Standard se lahko pohvali s popolno prenosljivostjo. Podprt je tako na UNIX, MacOS, kot tudi Windows operacijskih sistemih. Neodvisen pa je tudi od strojne opreme, saj deluje enako ne glede na tip ali proizvajalca grafične procesne enote.

6.1.1 Izvajalni model

OpenGL temelji na grafičnih primitivih. Med ukazi OpenGL ne boste našli ukazov za delo z okni ali opisovanje 3D predmetov. Sam standard OpenGL ne vsebuje niti ukazov za interakcijo z uporabnikom preko naprav, kot sta na primer tipkovnica in miš. Za to obstajajo določene knjižnice, ki razširjajo funkcionalnosti OpenGL.
OpenGL zahteva inicializacijo, ki je na primeru vizualizacije valovne enačbe predstavljena v naslednjem podpoglavju. V osnovi vizualizacija zajema kreiranje okna, definiranje atributov, ki vplivajo na izrisovanje (barvna shema, dvojni medpomnilnik, ...), v določenih primerih lahko v inicializaciji najdemo tudi parametre luči, kamere in nekaterih drugih objektov. Po inicializaciji program postavimo v glavno zanko, ki ima v grobem štiri korake. Na začetku počisti zaslon, sledi izris 3D scene, procesiranje vhodno izhodnih naprav in osvežitev okna.
Princip obnašanja OpenGL je do neke mere podoben obnašanju končnega avtomata. To pomeni, da ostane v nekem stanju, dokler stanja ne spremenimo. Primer je ukaz za aktivno barvo `glColor3f`. Barva, ki jo nastavimo, ostane aktivna, dokler ne na-

stavimo druge.

OpenGL programerju omogoča 3D programiranje na zelo nizkem nivoju. Programer se dejansko ukvarja z risanjem primitivov. Vsak izrisani objekt predstavlja množica točk, ki so povezane v primitive, ki se izrisujejo na zaslon. Za interaktivnost programer poskrbi s transformacijami teh objektov glede na signale vhodnih naprav, ki jih lahko upravlja uporabnik programa. OpenGL programerju sicer nudi več različnih načinov za predstavitev primitivov, še vedno pa se mora programer ukvarjati z izrisom primitivov in ne celotnih objektov.

Podlaga za kakršnokoli smiselno 3D aplikacijo je torej poznavanje osnovnih transformacij. Najbolj osnovne transformacije, ki jih lahko izvajamo nad točkami, so premik (translacija), skaliranje in rotacija. Matrike za posamezne transformacije lahko množimo v vrstnem redu *TRS* (translacija, rotacija, skaliranje) in tako dobimo končno matriko, s katero moramo pomnožiti točke, da dosežemo želeno transformacijo.

Za delo z matrikami OpenGL programerju nudi veliko uporabnih ukazov. Na voljo sta mu dva sklada matrik. Sklad *MODELVIEW* določa transformacije modela vključno s transformacijo pogleda, *PROJECTION* pa določa projekcijsko matriko. Programer lahko trenutno transformacijsko matriko shrani z ukazom *glPushMatrix* in jo prikliče nazaj z ukazom *glPopMatrix*.

Za bolj realen izris moramo v sceno dodati luči in posameznim objektom dodeliti lastnosti materiala. Za pravilno senčenje moramo poskrbeti, da so normale primitivov pravilno izračunane. Kjer nam material ni dovolj, lahko uporabimo teksture. Vključimo jih z ustreznim argumentom in ukazom `glEnable`. Teksture je potrebno kreirati, izbrati in jim določiti parametre izrisa. Pred izrisom vsakega oglišča lahko določimo koordinate teksture, ki bo mapirana na to oglišče.

V zahtevnejše aplikacije je potrebo dodati še na primer detekcijo trkov, sisteme delcev, z uporabo programskega OpenGL Shading Language pa lahko napišemo lastne programe za senčenje.

OpenGL programerju omogoča direktno upravljanje z grafično strojno opremo. Programiranje poteka na zelo nizkem nivoju, zato od programerja zahteva dobro poznavanje osnov računalniške grafike. Programiranje je nekoliko bolj zapleteno kot z uporabo grafičnih pogonov, ima pa programer veliko več svobode.

6.1.2 Izvedba

OpenGL sem izbral, ker programerju nudi zelo direkten dostop do grafične strojne opreme. Tako je mogoče implementirati deljenje grafičnega pomnilnika z OpenCL, ki ga potrebuje za izračun vsakega koraka. S takšnim programiranjem se izognemo stalnemu prenašanju podatkov iz pomnilnika grafične procesne enote v glavni pomnilnik računalnika in nazaj.
Za izris površine sem uporabil objekte tipa Vertex Buffer Object. Podatki o površini so tako ves čas v pomnilniku grafične procesne enote. Zato sem lahko uporabil deljenje pomnilnika med OpenGL in OpenCL. To prihrani veliko časa, saj med izračunom in izrisom ni potrebno prepisovati ali prestavljati podatkov. Pogoj za tako implementacijo je nekoliko drugačna inicializacija OpenCL.
Najprej je potrebno inicializirati OpenGL, kjer ustvarimo okno, definiramo določene atribute in določimo funkcije, ki skrbijo za izris in odziv na tipkanje po tipkovnici ter premikanje miške. Pred prvim izrisom z ukazom `gluPerspective` nastavimo še projekcijsko matriko in poskrbimo za ponastavitev *MODELVIEW* matrike. Po končani inicializaciji je potrebno kreirati VBO objekte z ukazom `glGenBuffers` in z ukazom `glBufferData` rezervirati prostor ter vanj prenesti začetne podatke. Ker OpenGL temelji na izrisu trikotnikov, je potrebno definirati še polje *Indices*, ki določa katere točke predstavljajo posamezen trikotnik.
Sledi inicializacija OpenCL, ki je precej podobna inicializaciji brez vizualizacije, vsebuje le nekaj manjših sprememb. Kontekst je potrebno ustvariti iz obstoječega OpenGL konteksta, kar omogoča, da tako OpenCL kot tudi OpenGL lahko dostopata do istega pomnilniškega prostora na grafični procesni enoti. Površino predstavimo z objektom VBO. Ustvarimo ga v inicializaciji OpenGL, v inicializaciji OpenCL pa z ukazom `clCreateFromGLBuffer` ustvarimo objekt *cl_mem*, ki kaže na isti prostor v pomnilniku grafične procesne enote.
Po končani inicializaciji OpenCL z ukazom `glutMainLoop` poskrbimo, da gre program v glavno zanko OpenGL, iz katere se ne vrne, dokler ne zahtevamo izhoda iz programa. Ko je program v glavni zanki, se vse dogaja preko povratnih klicev funkcij, ki smo jih definirali med inicializacijo.
Sam izračun je spremenjen le do te mere, da se rezultat posameznega izračuna zapiše v pravilno koordinato posamezne točke. Pred izračunom je potrebno določiti pravi-

len časovni korak, z ukazom `glFinish` poskrbeti, da je OpenGL končal z izrisom in z ukazom `clEnqueueAcquireGLObjects` pridobiti lastništvo nad *cl_mem* objektom. Po končanem izračunu je potrebno *cl_mem* objekte prepustiti OpenGL. To naredimo z ukazom `clEnqueueReleaseGLObjects`.

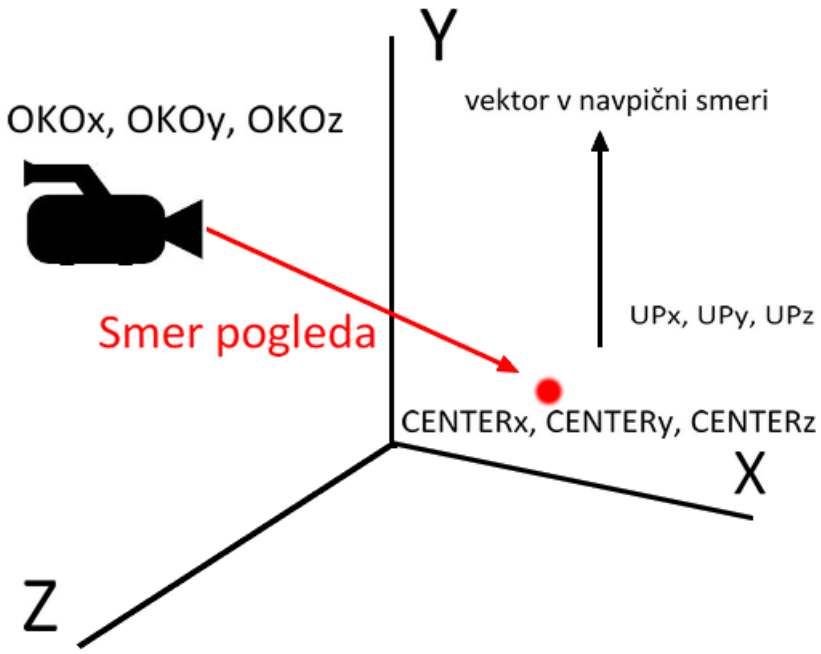
Izris temelji na trikotnikih, ki so določeni s poljem *Indices*. Podatke za posamezno točko OpenGL pridobi iz objektov VBO. Izris sprožimo z ukazom `glDrawElements`. Po klicu ukaza za izris poskrbimo za praznenje medpomnilnikov z ukazom `glFlush`, menjavo medpomnilnikov z ukazom `glSwapBuffers` in forsiranjem posodobitve okna z ukazom `glutPostRedisplay`.

Ker je bila vizualizacija namenjena čim hitrejšemu prikazu rezultatov izračunov enačbe, sem se odločil za izris žičnatega okvirja (angl. *wireframe*). S tem sem se izognil računanju normal trikotnikov in potrebi po osvetljevanju scene. Izris žičnatega okvirja sem dosegel z ukazom `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`.

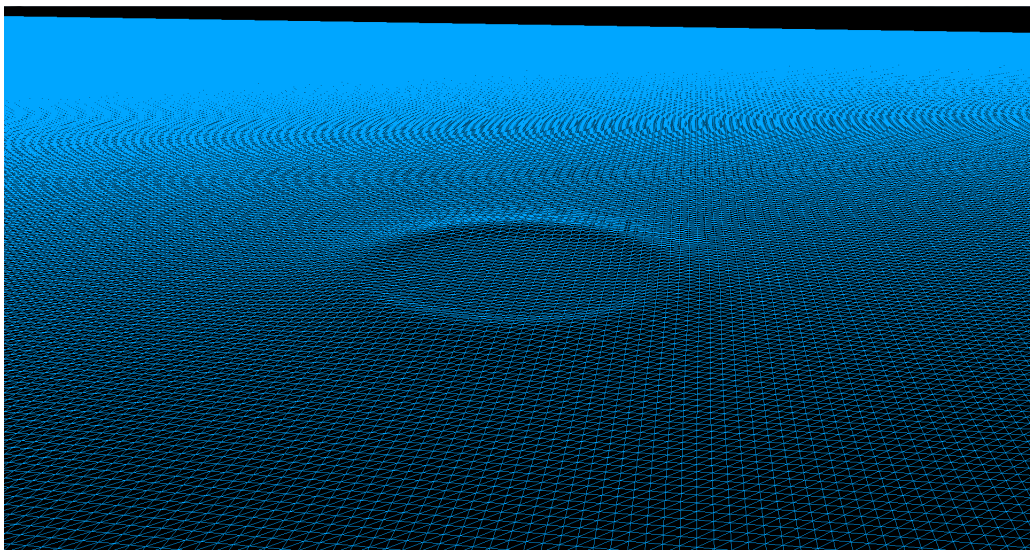
Za premikanje in rotiranje po sceni je potrebno implementirati določene transformacije. Ker je šlo v tem primeru za zelo osnovne transformacije enega objekta, sem to raje realiziral s premikanjem kamere. Za to poskrbi ukaz `gluLookAt`, ki sprejme 9 argumentov. Argumenti predstavljajo 3 trojice koordinat. Prva trojica opisuje položaj kamere ali očesa, druga trojica opisuje točko proti kateri gledamo, tretja trojica pa vektor, ki kaže v navpični smeri. Argumente ukaza `gluLookAt` grafično prikazuje slika 6.1.

Primer izrisa površine prikazuje slika 6.2.

Več o različnih načinih deljenja pomnilnika med OpenGL in OpenCL ter ustvarjanju deljenega konteksta si bralec lahko prebere v [17].



Slika 6.1: Ukaz `gluLookAt`



Slika 6.2: Izris površine po izračunu

Poglavje 7

Rezultati in primerjava CUDA ter OpenCL

Algoritem za numerično reševanje valovne diferencialne enačbe z Eulerjevo metodo in metodo Runge-Kutta 4. reda smo na način, ki je opisan v predhodnih poglavjih, implementirali na platformi CUDA in platformi OpenCL. Poskušali smo programirati tako, da sta si kodi na različnih platformah čim bolj podobni, saj bomo tako dobili najbolj objektivne rezultate. Zaradi precejšnih razlik med CUDA in OpenCL popolne enakosti seveda ne moremo doseči. Poskušali pa smo se ji čim bolj približati. Ob pregledu kode lahko hitro vidimo, da je koda OpenCL nekoliko zahtevnejša in tudi nekoliko daljša. To gre predvsem na račun drugačne zasnove platforme in strojne neodvisnosti. Glavne razlike v kodi lahko opazimo predvsem v inicializaciji, ščepca pa sta si med seboj zelo podobna.

7.1 Testno okolje

Ker objektivno primerjavo lahko dosežemo le s poganjanjem na identični strojni opremi, je bilo potrebno izbrati računalnik z grafično procesno enoto proizvajalca *nVidia*. Vsi testi so bili izvedeni na osebnem računalniku s procesorjem *Intel Core2Quad Q6600* z modificirano frekvenco jedra 3,2 Ghz, 8 GB DDR2 pomnilnika, s frekvenco 1066 Mhz in grafično procesno enoto *ZOTAC nVidia GeForce GTX760*.

Ostale komponente so izpuščene, saj bistveno ne vplivajo na hitrost izvajanja programa.
Program smo pognali za različno število korakov in različne velikosti matrik. Za vsako kombinacijo števila korakov in velikosti matrike iz tabele 7.1 sem program za računanje z Eulerjevo metodo in metodo Runge-Kutta 4. reda pognal trikrat ter kot rezultat uporabil povprečni čas.
V čas izračuna je vključena tudi rezervacija pomnilnika, kopiranje podatkov iz glavnega pomnilnika v pomnilnik grafične procesne enote in obratno ter sprostitev pomnilnika.

število korakov	velikost matrike
100	128x128
200	256x256
500	512x512
1000	1024 x 1024
2000	2048 x 2048
5000	4096 x 4096

Tabela 7.1: Število korakov in velikosti tabel za testiranje hitrosti programa

7.2 Rezultati

Vsi pridobljeni rezultati so predstavljeni v tabelah. Nekateri bolj zanimivi rezultati so prikazani tudi grafično. Zaradi lažje navedbe v tabelah je čas ponekod označen s kratico t.

7.2.1 Izračun

V spodnjih tabelah so prikazani rezultati meritev časa izračuna najprej za Eulerjevo metodo in nato še za metodo Runge-Kutta 4. reda. V posamezni tabeli je velikost matrike konstantna, število korakov pa se spreminja. Oznaka RK pri številu korakov

pomeni računanje z metodo Runge-Kutta 4. reda.

število korakov	čas CPE	čas CUDA	čas OpenCL	pohitritev CUDA	pohitritev OpenCL
100	0,032 s	0,005 s	0,006 s	6,40	5,33
200	0,077 s	0,011 s	0,011 s	7,00	7,00
500	0,168 s	0,026 s	0,028 s	6,46	6,00
1000	0,353 s	0,049 s	0,060 s	7,20	5,88
2000	0,539 s	0,095 s	0,106 s	5,67	5,08
5000	1,005 s	0,245 s	0,285 s	4,10	3,53
100 RK	0,177 s	0,020 s	0,024 s	8,85	7,38
200 RK	0,407 s	0,036 s	0,044 s	11,31	9,25
500 RK	0,968 s	0,093 s	0,114 s	10,41	8,49
1000 RK	1,823 s	0,193 s	0,235 s	9,45	7,76
2000 RK	2,944 s	0,414 s	0,451 s	7,11	6,53
5000 RK	5,298 s	1,036 s	1,162 s	5,11	4,56

Tabela 7.2: Izmerjeni časi pri velikosti matrike 128x128

Razlike v času izračuna na grafični procesni enoti v primerjavi s časom izračuna na cenralni procesni enoti so relativne majhne, saj ne presegajo faktorja 12. Temu je tako zaradi majhne matrike, kar pomeni manjše število podatkov, na katerih opravljamo izračun. Zato ne uspemo izkoristiti visoke stopnje paralelizma, ki nam ga nudi grafična procesna enota. Vidimo, da je CUDA v vseh primerih nekoliko hitrejša.

Pričakovali smo, da se bo z večanjem števila korakov večala tudi pohitritev, saj je inicializacija enaka ne glede na število korakov, med posameznimi koraki pa podatkov ne prenašamo nazaj v glavni pomnilnik naprave. Sam prenos podatkov v pomnilnik grafične procesne enote je porabil zelo malo časa, zato se to ne pozna veliko pri celotnem času izračuna.

Pri metodi Runge-Kutta 4. reda so pohitritve nekoliko večje, verjetno zaradi večje zahtevnosti izračuna. Razlike pa niso velike. Vzrok za to je verjetno sinhronizacija, saj se mora vsaka stopnja Runge-Kutta izračuna končati pred začetkom naslednje stopnje.

število korakov	čas CPE	čas CUDA	čas OpenCL	pohitritev CUDA	pohitritev OpenCL
100	0,083 s	0,007 s	0,008 s	11,86	10,38
200	0,191 s	0,013 s	0,011 s	14,69	17,36
500	0,491 s	0,034 s	0,035 s	14,44	14,03
1000	1,108 s	0,066 s	0,069 s	16,79	16,06
2000	2,285 s	0,124 s	0,137 s	18,48	16,68
5000	4,850s s	0,315 s	0,352 s	15,40	13,78
100 RK	0,511 s	0,025 s	0,030 s	20,44	17,03
200 RK	0,981 s	0,050 s	0,059 s	19,62	16,63
500 RK	2,415 s	0,131 s	0,149 s	18,44	16,21
1000 RK	5,708 s	0,253 s	0,290 s	22,56	19,68
2000 RK	12,115 s	0,559 s	0,592 s	21,67	20,46
5000 RK	27,320s s	1,446 s	1,440 s	18,89	18,97

Tabela 7.3: Izmerjeni časi pri velikosti matrike 256x256

Pohitritev se z večanjem velikosti matrike pričakovano povečuje. Pri velikosti matrike 256x256 pohitritev že presega vrednost 20.
CUDA je v veliki večini primerov zopet nekoliko hitrejša.
Metoda Runge-Kutta 4. reda dosega višje pohitritve, kar je pričakovano, saj je izračun bolj kompleksen.

število korakov	čas CPE	čas CUDA	čas OpenCL	pohitritev CUDA	pohitritev OpenCL
100	0,491 s	0,013 s	0,013 s	37,77	37,77
200	0,985 s	0,024 s	0,024 s	41,04	41,04
500	2,498 s	0,060 s	0,063 s	41,63	39,65
1000	5,015 s	0,117 s	0,118 s	42,86	42,50
2000	10,652 s	0,220 s	0,238 s	48,42	44,76
5000	28,351 s	0,563 s	0,602 s	50,36	47,09
100 RK	1,726 s	0,052 s	0,057 s	33,19	30,28
200 RK	2,645 s	0,106 s	0,089 s	24,95	29,72
500 RK	8,812 s	0,260 s	0,278 s	33,89	31,70
1000 RK	18,357 s	0,548 s	0,529 s	33,50	34,70
2000 RK	39,231 s	1,022 s	1,037 s	38,39	37,83
5000 RK	108,448 s	2,531 s	2,837 s	42,85	38,23

Tabela 7.4: Izmerjeni časi pri velikosti matrike 512x512

Pri velikosti 512x512 so pohitritve že kar velike. V primeru 5000 korakov Eulerjeve metode pohitritev presega številko 50. Zopet je CUDA v veliki večini primerov hitrejša.

Pri tej velikosti matrike so pohitritve pri metodi Runge-Kutta 4. reda manjši kot pri Eulerjevi metodi. Takih rezultatov nismo pričakovali, verjetno pa so posledica časa, ki ga porabi sinhronizacija. Pri metodi Runge-Kutta 4. reda nekaj več časa porabimo tudi za inicializacijo, saj moramo rezervirati prostor še za matrike, ki jih potrebujemo za vmesne izračune. Zaradi računanja vmesnih vrednosti porabimo tudi več predpomnilnika. Pri večji velikosti matrik je verjetno grafični procesni enoti že primanjkovalo predpomnilnika.

število korakov	čas CPE	čas CUDA	čas OpenCL	pohitritev CUDA	pohitritev OpenCL
100	1,861 s	0,032 s	0,033 s	58,16	56,39
200	3,780 s	0,065 s	0,060 s	58,15	63,00
500	9,485 s	0,162 s	0,161 s	58,55	58,91
1000	19,220 s	0,322 s	0,317 s	59,69	60,63
2000	40,152 s	0,652 s	0,602 s	61,58	66,70
5000	101,557 s	1,572 s	1,494 s	64,60	67,98
100 RK	6,561 s	0,155 s	0,158 s	42,33	41,53
200 RK	13,124 s	0,313 s	0,315 s	41,93	41,66
500 RK	33,526 s	0,820 s	0,726 s	40,89	46,18
1000 RK	66,934 s	1,569 s	1,442 s	42,66	46,42
2000 RK	136,229 s	2,881 s	2,880 s	47,29	47,30
5000 RK	339,422 s	7,662 s	7,998 s	44,30	42,43

Tabela 7.5: Izmerjeni časi pri velikosti matrike 1024x1024

Zopet vidimo, da se skladno s povečanjem matrike povečuje tudi pohitritev. Opazimo lahko, da je pri velikosti matrike 1024x1024 OpenCL že vsaj enakovreden konkurenčni platformi CUDA.
Metoda Runge-Kutta 4. reda zopet dosega nižje pohitritve, razlog je verjetno v pomanjkanju predpomnilnika.

število korakov	čas CPE	čas CUDA	čas OpenCL	pohitritev CUDA	pohitritev OpenCL
100	7,418 s	0,115 s	0,110 s	64,50	67,44
200	14,865 s	0,233 s	0,217 s	53,80	68,50
500	37,125 s	0,557 s	0,536 s	66,65	69,26
1000	74,498 s	1,092 s	1,051 s	68,22	70,88
2000	148,579 s	2,118 s	2,077 s	70,15	71,54
5000	374,909 s s	5,247 s	5,185 s	71,45	72,31
100 RK	25,914 s	0,532 s	0,533 s	48,71	48,62
200 RK	51,759 s	1,074 s	1,040 s	48,19	49,77
500 RK	129,661 s	2,583 s	2,547 s	50,20	50,91
1000 RK	259,716 s	5,171 s	5,090 s	50,23	51,02
2000 RK	508,431 s	9,044 s	9,285 s	56,22	54,76
5000 RK	1298,502 s	27,155 s	27,495 s	47,82	47,23

Tabela 7.6: Izmerjeni časi pri velikosti matrike 2048x2048

Pri velikosti matrike 2048x2048 se prvič zgodi, da je OpenCL v večini primerov hirejši od CUDA. Razlike niso velike, je pa opazno, da se OpenCL nekoliko bolje odreže pri večji količini podatkov. Pohitritve so že blizu maksimuma, ki ga pri reševanju tega problema lahko dosežemo s testno strojno opremo.
Razlike med Eulerjevo metodo in metodo Runge-Kutta 4. reda so vedno večje. Pohitritev pri metodi Runge-Kutta z večanjem velikosti matrike narašča zelo počasi.

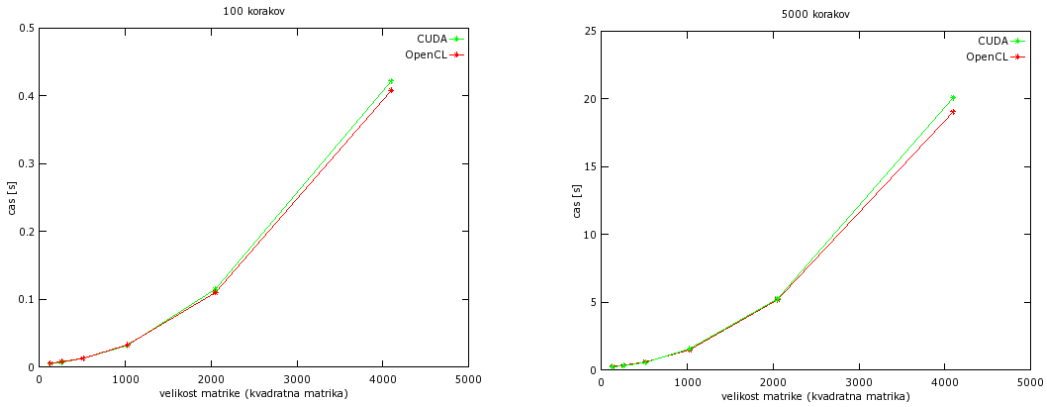
število korakov	čas CPE	čas CUDA	čas OpenCL	pohitritev CUDA	pohitritev OpenCL
100	29,758 s	0,422 s	0,408 s	70,52	72,94
200	59,421 s	0,841 s	0,801 s	70,66	74,18
500	148,446 s	2,088 s	1,981 s	71,10	74,93
1000	297,998 s	4,032 s	3,948 s	73,91	75,48
2000	589,547 s	8,056 s	7,884 s	73,18	74,78
5000	1477,133 s s	20,099 s	19,058 s	73,49	77,51
100 RK	102,556 s	1,979 s	1,970 s	51,82	52,06
200 RK	205,991 s	3,913 s	3,935 s	52,64	52,35
500 RK	510,024 s	9,671 s	9,764 s	52,74	52,24
1000 RK	1026,570 s	19,877 s	19,183 s	51,65	53,51
2000 RK	2089,151 s	37,841 s	37,913 s	55,21	55,10
5000 RK	5098,668 s	101,794 s	102,995 s	50,09	49,50

Tabela 7.7: Izmerjeni časi pri velikosti matrike 4096x4096

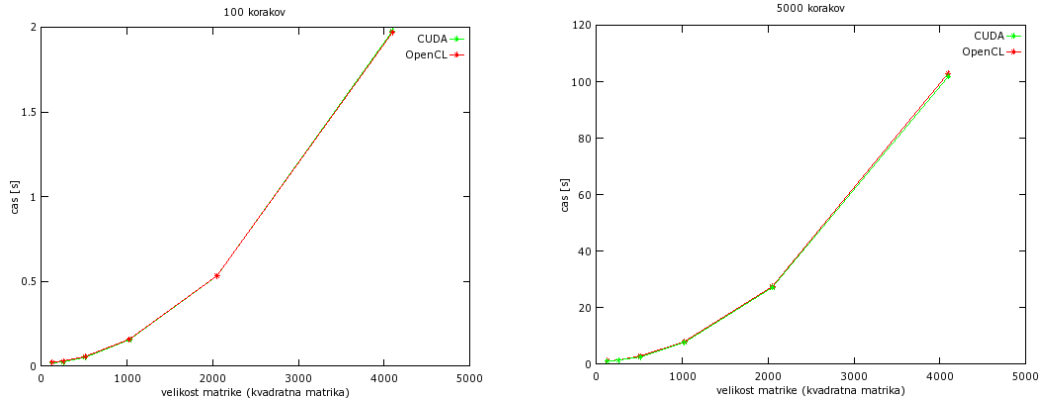
Pri največji testirani velikosti matrike dosegamo pohitritev glede na CPE do 78. Pričakovano se to zgodi pri največjem številu korakov in pri izračunu z OpenCL z Eulerjevo metodo. Z večanjem matrike se OpenCL hitro približuje rezultatom CUDA. Pri izračunu z Eulerjevo metodo je pri velikosti matrike 4096x4096 OpenCL vselej hitrejši od CUDA.
Pri metodi Runge-Kutta 4. reda sta platformi CUDA in OpenCL še bolj izenačeni, pohitritev pa segajo malo čez število 55.

V tabelah so rezultati grupirani glede na velikost matrike. Da si bralec lažje predstavlja kako na hitrost izračuna vpliva velikost matrike, smo izdelali grafe časa porabljenega za izračun v odvisnosti od velikosti matrike pri najmanjšem (100) in največjem (5000) številu korakov.

Iz grafov na sliki 7.1 je lepo razvidno, da je OpenCL izračun pri večjih matrikah



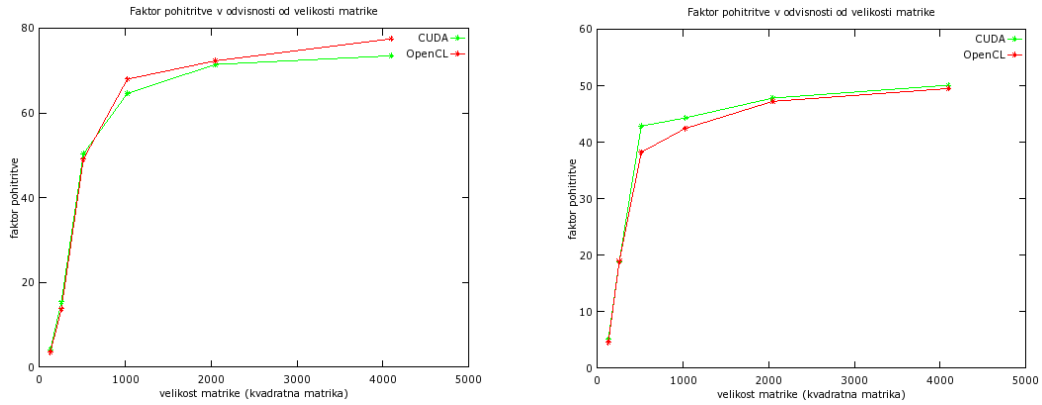
Slika 7.1: Graf časa porabljenega za izračun v odvisnosti od velikosti matrike pri 100 korakih (levo) in 5000 korakih (desno) z uporabo Eulerjeve metode.



Slika 7.2: Graf časa porabljenega za izračun v odvisnosti od velikosti matrike pri 100 korakih (levo) in 5000 korakih (desno) z uporabo metode RK4.

opravil nekoliko hitreje. Glede na absolutni čas izvajanja pa vidimo, da so razlike relativno majhne. Razlike med CUDA in OpenCL so na sliki 7.2 veliko manjše. V računanju valovne enačbe z metodo Runge-Kutta 4. reda sta si bila programa po času izvajanja izjemno blizu.

Grafa na sliki 7.3 prikazujeta pohitritve CUDA in OpenCL glede na CPE pri 5000 korakih in različnih velikostih matrike. Podatke za 5000 korakov smo izbrali zaradi največje razlike v pohitritvi napram izvajanju na centralni procesni enoti. Tako je odvisnost pohitritve od velikosti matrike najbolje razvidna.



Slika 7.3: Graf pohitritve glede na CPE v odvisnosti od velikosti matrike z uporabo Eulerjeve metode (levo) in metode RK4 (desno).

Vidimo, da krivulja pohitritve pri Eulerjevi metodi dlje časa strmo narašča. Pri matrikah velikosti več kot 1024x1024 pa obe krivulji naraščata veliko počasneje. Razlika v pohitritvi pri matrikah velikosti 2048x2048 in 4096x4096 je zelo majhna. Pri matrikah večjih od 4096x4096 lahko pričakujemo majhno povišanje pohtritve.

7.2.2 Vizualizacija

Pri testiranju časa, ki se porabi za računanje in izris skupaj, je bilo potrebno poskrbeti za neomejeno hitrost izrisovanja. Zaradi sinhronizacije osvežitve se hitrost izrisa pogosto omeji na hitrost osveževanja zaslona, zato je potrebno izklopiti sinhronizacijo *(angl. vertical sync)*.
Vizualizacijo smo pognali pri dveh različnih ločljivostih. Najprej pri manjši 1024x600 pik in kasneje še pri priljubljeni *Full HD* ločljivosti 1920x1080 pik.
Iz rezultatov je razvidno, da odločitev o platformi v primeru vizualizacije sploh ni ključna, saj so razlike tako majhne, da pri vizualizaciji praktično ne bi bile opazne.

velikost matrike	t/korak brez vizualizacije	t/korak 1024x600	pohitritev	t/korak 1920x1080	pohitritev
128x128	0,00006 s	0,0015 s	25	0,0015 s	25
256x256	0,00007 s	0,0017 s	24,29	0,0016 s	22,86
512x512	0,00014 s	0,0025 s	17,85	0,0023 s	16,43
1024x1024	0,00030 s	0,0055 s	18,33	0,0058 s	19,33
2048x2048	0,00104 s	0,0190 s	18,27	0,0195 s	18,75

Tabela 7.8: Izmerjeni časi za posamezni korak z vizualizacijo pri ločljivosti 1024x600 in 1920x1080 ter brez vizualizacije - Euler

V tabeli 7.8 so predstavljeni rezultati meritev porabljenega časa z vizualizacijo in brez pri uporabi Eulerjeve metode.
Vizualizacija v primerjavi z izračunom porabi ogromno časa. Razmerje med časom porabljenim za izračun z vizualizacijo in za sam izračun se z večanjem velikosti matrike manjša. Vzrok za to je v inicializaciji izrisa, ki očitno potrebuje relativno veliko časa. V inicializaciji so zajete transformacije in priprava VBO objektov za izris. Sam izris je zelo hiter, zato večja velikost matrike pri izrisu ne zahteva veliko dodatnega časa.
Tudi ločljivost zelo malo vpliva na čas vizualizacije. Vzrok za to se verjetno skriva v preprostosti scene, ki jo izrisujemo. Razlika bi bila verjetno bolj vidna šele ob bolj

drastičnem povečanju ločljivosti, kar pa je, glede na ločljivosti večine zaslonov, ki jih dandanes uporabljamo, nesmiselno.

velikost matrike	t/korak brez vizualizacije	t/korak 1024x600	pohitritev	t/korak 1920x1080	pohitritev
128x128	0,00023 s	0,0015 s	6,52	0,0014 s	6,09
256x256	0,00029 s	0,0018 s	6,21	0,0017 s	5,86
512x512	0,00057 s	0,0027 s	4,74	0,0033 s	5,79
1024x1024	0,00160 s	0,0075 s	4,69	0,0081 s	5,06
2048x2048	0,00560 s	0,0268 s	4,79	0,0270 s	4,82

Tabela 7.9: Izmerjeni časi za posamezni korak z vizualizacijo pri ločljivosti 1024x600 in 1920x1080 ter brez vizualizacije - RK4

V tabeli 7.9 so predstavljeni rezultati meritev porabljenega časa z vizualizacijo in brez pri uporabi metode Runge-Kutta 4. reda.
Vidimo, da je v tem primeru razlika med izvajalnim časom z vizualizacijo in brez manjša. To je pričakovano, saj izračun potrebuje veliko več časa, kot pri Eulerjevi metodi, vizualizacija pa je identična. Če v rezultatih izračunov preverimo čase izračuna z Eulerjevo metodo in metodo Runge-Kutta 4. reda vidimo, da se razlikujejo približno za faktor 5. Za podoben faktor se razlikujejo tudi razlike med časom porabljenim za izračun in vizualizacijo ter časom porabljenim le za izračun pri uporabi Eulerjeve metode in metode Runge-Kutta 4. reda.

Poglavje 8

Zaključek

V zadnjem desetletju se je paralelno programiranje zelo razvilo in razširilo. Vzrok za to lahko najdemo v dejstvu, da so dandanes skoraj vse centralne procesne enote večjedrne. Proizvajalci ne povečujejo same frekvence ure procesne enote, ampak na isti čip namestijo več procesnih enot, ki lahko simultano izvajajo operacije. Da lahko programer izkoristi tako arhitekturo centralnih procesnih enot, mora poznati paralelno programiranje.

Skladno z razvojem paralelnega programiranja in z manjšim zamikom se je razvijalo tudi programiranje na grafični procesni enoti. Osnovna ideja je podobna paralelnemu programiranju, stopnja paralelizma pa je veliko večja. Grafične procesne enote so doživele velik napredek in v času pisanja imajo nekatere že več tisoč jeder. Kljub temu, da so jedra na grafični procesni enoti veliko bolj primitivna od jeder na centralni procesni enoti, lahko zaradi njihove velike količine dosegamo zavidljive pohitritve. Veliko pa je odvisnega od programerja, saj mora čim bolje izkoristiti veliko število jeder.

Pred začetkom programiranja se vsak programer sreča pred pomembno odločitvijo. Izbrati mora platformo za programiranje na grafični procesni enoti. V diplomskem delu smo predstavili platformi CUDA in OpenCL ter ju na problemu valovanja vode primerjali. Algoritem, ki teče na grafični procesni enoti, v primerjavi z algoritmom za centralno procesno enoto dosega več kot 7700% pohitritve. Primerjava platform je pokazala, da sta si po zmogljivosti zelo podobni. CUDA se je nekoliko bolje od-

rezala pri kopiranju podatkov v pomnilnik grafične procesne enote in je bila zaradi tega hitrejša pri manjših matrikah, kjer je izračun zelo hiter. OpenCL je bil boljši pri izračunih na večjih matrikah. Implementirali smo tudi vizualizacijo površine z uporabo standarda OpenGL. Da smo dosegli hitrejše izvajanje programa, smo uporabili deljenje pomnilnika med OpenCL in OpenGL. Meritve so pokazale, da vizualizacija v primerjavi z izračunom porabi zelo veliko časa. Zato izbira platforme v primeru vizualizacije ni ključnega pomena.

Programiranje na grafični procesni enoti nam je pri reševanju določenih problemov v veliko pomoč, saj lahko probleme rešujemo hitreje kot na centralni procesni enoti. Programer pa mora poznati osnovno arhitekturo grafične procesne enote, osnove paralelnega programiranja in sintakso jezika, ki ga bo uporabil za programiranje na grafični procesni enoti. Zaradi dodatnega dela, ki ga ima s tem programer, so se že pojavili prototipi visoko nivojskih programskih jezikov za programiranje na grafični procesni enoti [18], ki programerju olajšajo programiranje.

V prihodnosti lahko pričakujemo vedno večje zanimanje za programiranje na grafični procesni enoti. Dandanes se grafične procesne enote uporabljajo za reševanje najrazličnejših problemov. Zaradi strukture problema so zelo zanimivi problemi procesiranja slik [19], [20]. Z nadaljnim razvojem grafičnih procesnih enot se bo spekter teh problemov verjetno še povečal.

Slike

Literatura

[1] S. R.R., "Moore's law: past, present and future," *Spectrum, IEEE*, vol. 34, no. 6, pp. 52–59, 1997.

[2] J. Fang, A. Varbanescu, and H. Sips, "A comprehensive performance comparison of cuda and opencl," in *Parallel Processing (ICPP), 2011 International Conference on*, 2011, pp. 216–225.

[3] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era." *IEEE Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[4] Y. Shi, "Reevaluating amdahl's law and gustafson's law," *Computer Sciences Department, Temple University (MS: 38-24)*, 1996.

[5] B. Orel, *Osnove numerične matematike.* Fakulteta za računalništvo in informatiko, 2004.

[6] nVidia. (2014, Sep) Cuda toolkit documentation. http://docs.nvidia.com/cuda/.

[7] IEEE. (2014, Sep) Ieee 754: Standard for binary floating-point arithmetic. http://grouper.ieee.org/groups/754/.

[8] D. Kodek, *Arhitektura in organizacija računalniških sistemov.* BI-TIM, 2008.

[9] J. Nickolls and W. Dally, "The gpu computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, 2010.

[10] R. Farber, *CUDA application design and development.* Elsevier, 2011.

[11] G. M., L. G. S., N. J., A. J., H. J., M. S., P. E., Y. Zhang, and V. V., "Parallel computing experiences with cuda," *Micro, IEEE*, vol. 28, no. 4, pp. 13–27, 2008.

[12] K. Group. (2014, Sep) Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[13] J. Shen, J. Fang, H. Sips, and A. Varbanescu, "Performance traps in opencl for cpus," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, 2013, pp. 38–45.

[14] Q. Lan, C. Xun, M. Wen, H. Su, L. Liu, and C. Zhang, "Improving performance of gpu specific opencl program on cpus," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012 13th International Conference on*, 2012, pp. 356–360.

[15] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[16] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*. Addison-Wesley, 2011.

[17] Intel. (2014, Sep) Opencl and opengl interoperability tutorial. https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial/.

[18] H. T. . A. T.S., "hicuda: High-level gpugpu programming," *Parallel and Distributed Systems, IEEE*, vol. 22, no. 1, pp. 78–90, 2011.

[19] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. Kim, "Design and performance evaluation of image processing algorithms on gpus," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 91–104, 2011.

[20] W. Wang, Y. Zhang, S. Yan, Y. Zhang, and H. Jia, "Parallelization and performance optimization on face detection algorithm with opencl: A case study," *Tsinghua Science and Technology*, vol. 17, no. 3, pp. 287–295, 2012.